

# ČIPY, DATA, PROCESORY

## Počítače na koleni II

(C) Martin Malý, 2019  
CC-BY-NC-ND

## *Předmluva*

---

### *ToDo*



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

**Podpoříte její vznik?**

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na  
<https://www.osmibity.cz/addons.html>

# Krystalové oscilátory

Na to, že je přesný hodinový zdroj srdcem každého počítače, jsme se vlastními oscilátory moc nezabývali. Nemuseli jsme – všechny procesory, které jsme použili, měly zabudovaný oscilátor a stačilo jen připojit krystal a zátěžovou kapacitu (load capacity). Zátěžová kapacita má nejčastěji podobu jednoho nebo dvou kondenzátorů s kapacitou mezi 22 a 48 pF, a když nevíte, jakou přesně použít, podívejte se do datasheetu k procesoru, a tam to bývá popsáno.

Důležité je jen dodržet pravidlo „velká zemnicí plocha kolem krystalu“. Při frekvencích, které používáme, nejsou kapacity, včetně těch parazitních, jako třeba kapacita plošného spoje, nijak kritické. U rychlejších obvodů by to byl problém, ale okolo 2 – 4 MHz, kde se pohybujeme my, se není čeho bát.

Když se do teorie krystalových oscilátorů ponoříte hlouběji, zjistíte, že možných zapojení je několik a že se rozlišují dva základní typy: paralelní a sériový. Krystaly mohou kmitat na dvou základních frekvencích, které se, aby se to nepletlo, rovněž nazývají sériová a paralelní. Sériová je o něco málo nižší než paralelní.

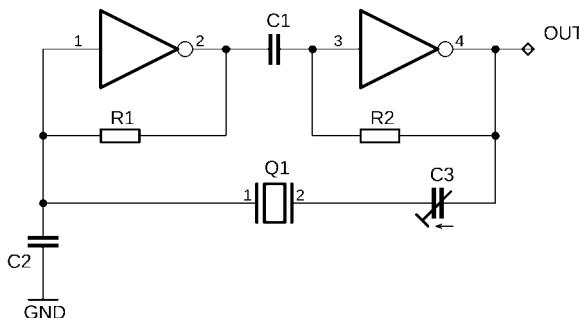
Obecně platí, že krystal se chová jako součástka, jejíž impedance (odpor) je nejnižší, pokud jí prochází střídavý proud s rezonanční frekvencí rovnou jeho mechanickým vlastnostem. Impedance krystalu je nulová ve dvou bodech – v rezonančním bodu (sériová frekvence) a v antirezonančním bodu (ideální paralelní frekvence).

U krystalů, které používáme my, tedy s kmitočty v řádu jednotek megahertzů, většinou není nutné řešit přesný rozdíl mezi sériovou a paralelní frekvencí. Stejně tak pro osmibitový hobby počítač není moc potřeba řešit teplotní kompenzaci a podobné věci, které musí řešit návrháři velmi přesných hodin.

Pokud si budete stavět samostatný krystalový oscilátor pro číslicová zařízení, pravděpodobně zvolíte zapojení s invertory. Ostatně i v těch mikroprocesorech je použité stejné zapojení. Pomocí jednoho (paralelní) nebo dvou (sériové) invertorů vytvoříte první část oscilátoru, totiž zesilovač. O druhou část oscilátoru, o zpětnou vazbu, se postará právě krystal.

První zapojení, sériové, používá dva invertory, mezi nimiž je zapojen vazební kondenzátor C1 (někdy bývá vynechán) s kapacitou jednotek či desítek nF. Jeho úkolem je odfiltrovat stejnosměrný proud. Oba invertory mají zapojené zpětnovazební rezistory, které je udržují v oblasti lineárního zesílení. Jejich hodnota není kritická, pro frekvence 1 – 4 MHz se

doporučuje okolo 2k2. Jiné zdroje doporučují R2 spočítat jako  $3000/f$ , kde f je frekvence v MHz.

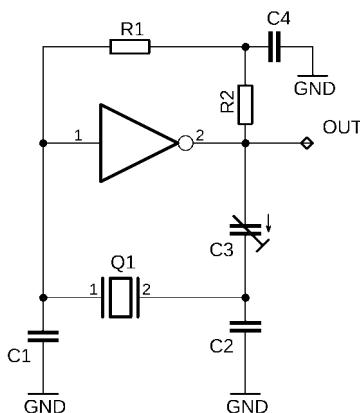


Kondenzátor C2 je již zmíněný zátěžový – většina krystalů pro naše frekvence počítá se zátěžovou kapacitou mezi 22 – 33 pF, ale jak už jsem psal: hodnota není zcela kritická.

Raději to explicitně zmíním: Když píšu, že „hodnota není kritická“, tak tím mám na mysli, že není kritická pro uvažované použití, tedy generátor kmitů v řádu jednotek MHz bez nároku na vysokou přesnost.

Pomocí trimru C3 můžete kmitočet celého zapojení ještě jemně doladit, ale pokud nepotřebujete vysokou přesnost, můžete trimr vynechat.

Paralelní oscilátor používá pouze jeden jediný invertor.



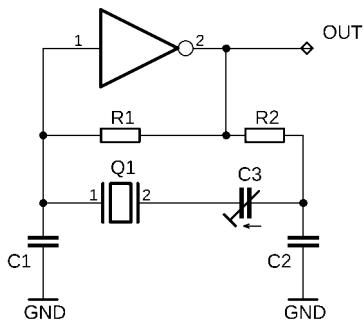
Rezistory R1, R2 a kondenzátor C4 fungují jako zpětná vazba pro udržení invertoru v oblasti lineárního zesílení. R1 a R2 mají odpor 1k, kondenzátor C4 kapacitu okolo 250 nF.

Zátěžové kondenzátory C1 a C2 mají opět kapacitu 22 – 33 pF, trimr C3 dolaďuje frekvenci, není nutný.

Pokud si postavíte tento oscilátor a použijete invertor 74HC04, 74HCT04 nebo jiný CMOS oscilátor, zjistíte, že si ani nekmitnete. Oscilátor prostě nepoběží a nebude kmitat.

Někde se můžete setkat s názorem, že to je proto, že CMOS obvody jsou příliš rychlé, ale to není ten pravý důvod. Ony ve skutečnosti nejsou výrazně rychlejší než TTL LS. Pravý důvod je ten, že mají vyšší zisk a obrovský vstupní odpor. To znamená, že ve výše uvedeném zapojení je velmi těžké udržet je v lineární oblasti.

Proto se používá modifikované zapojení s rezistorem sériově zapojeným u krystalu.



Rezistor R1, který udržuje invertor v pracovní oblasti, může mít řádově stovky kilohommů. Typicky se používá hodnota 180 k, která je pro CMOS invertory více než dostatečná. Ale můžete použít téměř cokoli od 47 k po 1 M.

Ladicí trimr C3 se opět používá k jemnému doladění, nebo se vynechává. Rezistor R2 pak omezuje proud krystalem, ale zároveň spolu s kondenzátorem C2 tvoří dolní propust, která brání krystalu kmitat na vyšších harmonických frekvencích.

Pokud v tomto zapojení použijete TTL hradlo, třeba 74LS04, nebude kmitat. Velmi velký odpor ve zpětné vazbě jej nedokáže udržet v pracovní oblasti.

Výstup oscilátorů se většinou nepoužívá přímo. Ačkoli jsou invertory součástky číslicové, v těchto zapojeních z nich děláme trochu analogové zesilovače a nutíme je pracovat v oblastech zakázaného pásmo. Kondenzátory

navíc deformují strmost hran. Bývá proto dobrým zvykem na výstup připojit ještě jeden invertor, který ošetří strmost hran, posílí výstup a oddělí jej od zbytku obvodu.

- <https://www.electronics-tutorials.ws/oscillator/crystal.html>
- <https://www.analog.com/media/en/technical-documentation/application-notes/an12fa.pdf>
- <https://www.changpuak.ch/electronics/Oscillators.php>
- <https://www.ecsxtal.com/store/pdf/Oscillation-Circuit-Design-Considerations.pdf>
- <https://www.eleccircuit.com/simple-crystal-oscillator-circuit/>
- [https://www.eit.lth.se/fileadmin/eit/courses/edi021/PDF\\_files/oscillators.pdf](https://www.eit.lth.se/fileadmin/eit/courses/edi021/PDF_files/oscillators.pdf)
- <http://www.z80.info/uexosc.htm>



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

Podpoříte její vznik?

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na <https://www.osmibity.cz/addons.html>

# Oživení počítače

Bravo:

Zapájení patic, kondenzátorů, rezistorů, tlačítka, krystalů a napájecích přívodů

Test spojení: GND na paticích, VCC na paticích, test, zda není zkrat gnd/vcc

Test spojení PHI2: pin 39 CPU / pin 27 ACIA

Test hodin: Zasunout CPU, zapnout napájení, měřit f mezi GND a pin 39 CPU – frekvence 3.6864 MHz, napětí cca 2.9 V

Zasunout paměti RAM, EEPROM a 7400. Spustit

Monitor by měl čekat ve smyčce, takže proměřit vývody A0, A1, A2, A3, A4 a změřit frekvenci. A0 je cca 1.6 MHz.

Logické úrovně na EEPROM od vývodu 1 (A14):

Vstup	Signál	Úroveň	f (kHz)
1	A14	--- (NC)	
2	A12	H	
3	A7	H	
4	A6	H	
5	A5	H	
6	A4	1,5 V	720
7	A3	4,6 V	195
8	A2	2,8 V	1000
9	A1	3,1 V	745
10	A0	2,7 V	1550

Zapojit dekodér 74138.

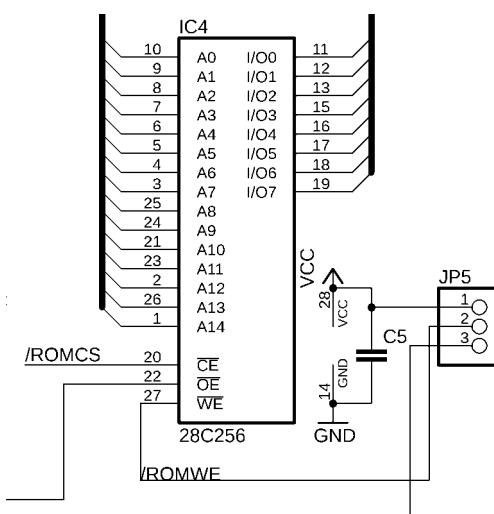
Sám sobě programátorem  
EEPROM

U všech konstrukcí OMEN jsem v zapojení paměti EEPROM navrhl jumper / přepínač, většinou nazvaný WREN – WRite ENable, a zároveň jsem psal, že při práci by měl být zapojený tak, aby na vstupu /WE byla stále logická 1.

Tak proč tam tedy je? Proč jsem ho nepřipojil přímo na +5 voltů?

Když je vstup /WE u paměti EEPROM připojený na logickou 1, je zakázaný zápis. A to je dobré, tak to má být a tak to chceme. Většinou chceme z paměti (EEP)ROM jen číst. Pokud do ní nějaký kód zapisuje, je to většinou omylem, a kdyby se zapisovat mohlo, přineslo by to spíš mrzení, než užitek.

Ale existují i situace, kdy je dobré zápis povolit. Například tehdy, kdy chcete aktualizovat firmware, obslužný program, nebo do EEPROM přidat nějaké rozšíření. Což jsou věci, které čas od času mohou být k užitku, proto je dobré je nějakým způsobem umožnit, ale zároveň je člověk nepoužívá denně a rutinně, tak nevadí, když jejich povolení je trochu netriviální. Pokud možno tak složité, aby k němu nemohlo dojít náhodou. Ideální je proto použít takzvaný pin header, neboť tři kovové kolíčky, a věc, které se říká „shunt“ nebo též „jumper“, a která vodivě spojí vývody 1-2, nebo 2-3.



Na této ukázce ze schématu OMEN Alpha je zmíněný přepínač zakreslený s označením JP5. Pokud jsou spojené vývody 1 a 2, je na vstup /WE přivedeno napájecí napětí. Pokud jsou spojené vývody 2 a 3, je na tento vstup přiveden signál /WR od procesoru. Podobné zapojení je i u Brava a Echa, protože princip práce je stejný.

Asi tušíte, že v zápisu do paměti EEPROM bude nějaký háček, a tušíte správně. Asi ten hlavní háček, přímo hák, je ten, že se něco nepovede. Data se zapíšou špatně, program bude mít chyby, přepíšete si základní programové vybavení, a po restartu budete mít zařízení ve stavu „brick“, tedy funkční obdobu cihly.

Druhý háček vyplývá ze samotného principu zápisu dat do paměti EEPROM. Z hlediska připojení se taková paměť neliší od RAM: přivedete data na datovou sběrnici, adresu na adresní, aktivujete signály /CS a /WE, a je to! Jenže EEPROM je přeci jen primárně ROM, tedy paměť pouze ke čtení, a ačkoli máme možnost přepsat data, má to své závludnosti. Třeba to, že zápis trvá docela dlouho, a po tu dobu by paměť neměla být rušena, a už vůbec by po ní neměl někdo chtít nějaká data.

Většina pamětí EEPROM, a platí to i pro oba typy, co v konstrukcích používáme, tedy AT28C64 a AT28C256, umí zapsat data buď po jednotlivých bajtech, nebo po celých blocích, ale poté, co zapíšeme data, respektive celý blok, musíme čekat. U těchto pamětí je čekací doba až 10 milisekund; u jejich rychlejších variant, třeba 28C256F, to jsou 3 milisekundy.

To fakt není žádná super rychlosť. Kdybychom zapisovali bajt po bajtu do paměti OMEN Alpha, což je 32 kB, tak to bude trvat  $32768 \times 10 \text{ ms} = 327,68 \text{ s}$ , což je něco přes pět minut. Pokud využijeme zápisu po blocích 64 bytů, zvládneme to za pět sekund.

Důsledkem tohoto omezení je, že program, který zapisuje do EEPROM, nemůže být v této EEPROM zároveň spuštěný. Po zápisu bajtu by procesor přistupoval k téže paměti. Paměť by to brala jako znamení, že má začít zapisovat, na 10 milisekund by se odmlčela, ale procesor by během těch deseti milisekund z téže paměti četl instrukce. Jenže po dobu zápisu paměť posílá na datovou sběrnici informace o tom, že zapisuje, resp. že už zápis skončil, takže by se místo instrukcí četly nesmysly, program by okamžitě zhavaroval, a *brick* by nastal dřív, než stačíte říct „RESET!“

Takže dvě zlatá pravidla:

1. Program, který zapisuje do EEPROM, musí celý kompletně běžet v RAM a musí zajistit, že během zápisového cyklu se na sběrnici neobjeví žádná adresa, která by aktivovala paměť EEPROM.
2. Program, který přepisuje EEPROM, musí být trojnásob opatrný než běžný program. Všechna data zkонтrolujte dřív, než je prohlásite za zapsaná, raději pomaleji, ale bezpečně, nezapomeňte zakázat všechna přerušení a modlete se, aby odněkud nepřišlo nemaskovatelné přerušení.

## Jak zapsat do EEPROM?

Samotný zápis je úplně stejný jako při zápisu do RAM. Použijte jakoukoli instrukci, která zapisuje do paměti, a pokud bude adresa odpovídat EEPROM a předtím povolíte zápis do EEPROM (hardwarem přepínačem, viz výše), paměť začne ukládat zapsaný bajt na zadanou adresu. Během zápisu z paměti nelze číst data, namísto toho načítáte stavové informace.

Blokový zápis umožňuje zapsat až 64 po sobě jdoucích bajtů najednou. Pro zápis bloku není potřeba žádná speciální instrukce, prostě zapisujete na adresu a, a+1, a+2, ... Stačí dodržet dvě prostá pravidla:

- Mezi zapsanými daty nesmí uplynout více než 150 mikrosekund, což není většinou problém; u Alphy to vychází na 276 hodinových taktů, během té doby stačíme spolehlivě připravit nový bajt a zapsat ho.
- Všechny adresy v bloku musejí mít stejnou hodnotu v bitech A6 – A14 (u 28C256; u paměti 28C64 to jsou bity A6 – A12). Při zápisu bloků můžete měnit bity A0 – A5, tedy nejnižších 6 bitů adresy.

Jakmile tato pravidla nedodržíte, tj. nastane větší časová prodleva nebo další adresa překročí hranici bloku 64 bajtů, paměť další data zahodí a spustí interní zápisový cyklus.

Po spuštění interního zápisového cyklu paměť další požadavky na zápis ignoruje. Tento cyklus může trvat až 10 milisekund, jak jsme si už řekli. Během této doby paměť informuje o tom, že se zapisují data, prostřednictvím dvou mechanismů, /DATA Polling a Toggle bit.

/DATA Polling spočívá v tom, že program čte data ze stejné adresy, kam naposledy zapisoval. Dokud trvá zápis, je v nejvyšším bitu D7 invertovaná hodnota zapisované hodnoty. Tedy pokud jste jako poslední hodnotu zapsali například 0x55 (což je binárně 01010101) na adresu 0xAAA, budete při čtení z téže adresy načítat hodnotu, která má v nejvyšším bitu 1. Jakmile cyklus zápisu skončí, bude načtená hodnota odpovídat tomu, co jste do paměti zapsali.

Druhá metoda kontroly je bit toggling – během cyklu zápisu se při čtení bude hodnota bitu D6 pravidelně měnit 0 – 1 – 0 – 1 atd. Když cyklus zápisu skončí, hodnota D6 se měnit přestane.

Obě tyto metody můžete použít k určení, zda je bezpečné zapisovat další data. Alternativní postup je prostě vyčkat deklarovaných 10 milisekund. Ale reálná kontrola stavu pomocí technik /DATA polling nebo toggle bit je nejjistější.

Můžete se inspirovat malým ukázkovým programem (pro OMEN Alpha), kde je použita právě technika /DATA polling:

<https://8bt.cz/eepw>

---

# FPGA

Trocha historie nikoho nezabije, na rozdíl od sestavování složitých kombinačních obvodů...

No dobře, přeháním, ani sestavování kombinačních obvodů není letální, ale představte si takový dekodér, jaký jsme měli v počítačích OMEN. Máte? Tak, teď si jej představte o něco složitější, ne s pouhými dvěma vstupy. Představte si jemnější škálování, třeba po 1 kb blocích, to máte šest adresních vstupů, a představte si složitější proklady, kdy potřebujete vygenerovat různé kombinace povolovacích signálů...

Samozřejmě že lze nakreslit pravdivostní tabulku (spíš tabuli), Karnaughovu mapu, sepsat logické výrazy a pokoušet se to namapovat do NANDů, NORů, třívstupových, čtyřvstupových, osmivstupových hradel a různých AND-OR-INVERTů, a nakonec skončíte s něčím, co zabírá pět integrovaných obvodů, některé použité třeba jen z poloviny. Funguje to, to ano, ale topí to a zabírá to spoustu místa.

## *Programovatelné obvody*

---

### **PROM**

---

V letech dávno minulých se podobné problémy, pokud jste na to měli příslušné vybavení, řešily pomocí paměti PROM. Abychom si rozuměli: Opravdu platí, že PROM jsou programovatelné paměti ROM, a že by v nich měly být nějaké hodnoty konstant a tak, ale když to vezmete kolem a kolem, tak takový kombinační logický výraz můžeme zapsat do tabulky, spočítat jeho hodnoty pro všechny možné vstupní kombinace, a pak výsledky naprogramovat do paměti PROM. Jednou zapsaná data zůstanou zapsaná navždy, tak co by ne?

Oblíbená paměť PROM byla typu 74188 / 74288. Má organizaci 32x8, tedy 32 slov po 8 bitech. Jinými slovy má pět adresních vstupů a osm datových výstupů, takže s ní hravě pokryjete případy kombinačních obvodů, které mají do pěti vstupů a do osmi výstupů.

Druhá oblíbená paměť byla 74287 s organizací 256x4. Tedy osm vstupů, čtyři výstupy.

Když jste se podívali v osmdesátých letech do Amatérského radia na číslicové konstrukce, byla taková paměť PROM často i na místech, kde by si autor vystačil s dvěma pouzdry. Asi bylo leckdy jednodušší použít paměť

PROM. Mám ale takové podezření, že to hodně záleželo na tom, zda dotyčný návrhář měl přístup k těmto obvodům a k programátoru, nebo naopak zda jeho šuplík oplýval spíš obvody TTL SSI a MSI...

A tak se obvody 188 a 287 objevovaly v rolích dekodérů a složitých kombinačních obvodů mezi procesorem a jeho periferiemi (vžil se název „glue logic“), až do doby, než někoho napadlo, že by to šlo jinak.

### ***PLA***

---

Programmable Logic Array, ve zkratce PLA a česky Programovatelné logické pole, je součástka, která přišla na trh za tím účelem, který jsme si právě popsali: vytvořit složitý kombinační obvod v jednom pouzdro. Na rozdíl od PROM, kde se stylem „brute force“ spočítaly hodnoty pro všechny možné kombinace a ty se zapsaly do paměti, u PLA byl návrh bližší tomu obvodovému.

PLA si můžeme představit jako sestavu „pole AND“, „pole OR“ a „pole INVERT“. Každý z těchto bloků je zapojený jako matice N sloupců (vstupy) a M řádků (logická hradla). Podle toho, které propojky naprogramujeme (podobně jako u PROM), takovou funkci na výstupu dostaneme.

Podobnou funkci měly obvody PAL (Programmable Array Logic). Pomocí propojek („fuses“) se při programování určí, které vstupní signály mají vést do jakého bloku AND\_OR\_INVERT. Obvody PAL se postupně vyvinuly, podobně jako paměti, nejprve do podoby mazatelné UV světlem (PALC) a posléze do elektricky přeprogramovatelných obvodů (PALCE).

Výrobci začali do obvodů přidávat i složitější celky. Například možnost mít u výstupů registr nebo signál na výstupu dále zpracovávat.

Obvody PAL se programovaly podobně jako PROM. Technicky vlastně o PROM / EPROM šlo. Aby nebylo nutné ručně počítat, které propojky se mají nastavit a které nechat, existovaly nástroje jako ABEL, CUPL nebo PALASM, které dokázaly zpracovat logické výrazy zapsané v nějaké formalizované podobě a z nich připravit výstup, vhodný k programování obvodů (nejčastěji ve formátu JEDEC).

Společnost Lattice představila v roce 1983 další vylepšení obvodů PAL s názvem GAL – Generic Array Logic. Tyto obvody byly vývodově kompatibilní s obvody PAL, ale šlo je jednodušeji přeprogramovat, některé z nich i v hotovém zařízení („in place“ nebo „in circuit“).

## *CPLD*

Obvody PAL a GAL dokázaly nahradit několik obvodů SSI, MSI. Jejich nástupci, obvody CPLD (Complex Programmable Logic Devices), nahradily několik tisíc hradel. Novější generace až stovky tisíc hradel.

Obvody CPLD mají s předchozími generacemi programovatelných obvodů společný princip zaznamenávání konfigurace do interní paměti (EE)PROM, a mnohé mají naopak přiřazené určité vnitřní bloky ke konkrétním pinům.

Hlavní rozdíl je ale ten, že CPLD obsahují řádově více vnitřních bloků a mají mnohem komplexnější možnosti vnitřního propojení těchto bloků.

Vnitřní bloky jsou rovněž mnohem bohatší než u PAL/GAL. Buňka (macrocell) se typicky skládá z klopného obvodu / registru (představme si ho jako klopný obvod D s nastavením a nulováním, jako je v obvodu 7474), konfigurovatelné logické sítě na jeho vstupech a konfigurovatelných multiplexorech na výstupech.

Například u oblíbených CPLD řady XC9500 od společnosti Xilinx udává poslední dvojice či trojice číslic označení počet těchto buněk. Například XC9572 jich má 72. V této rodině máte na výběr mezi 36, 72, 108, 144, 216 a 288 makrobuňkami. Největší zástupce této řady, XC95288, nabízí zároveň 6400 hradel k použití.

Řada XC9500 používá pro konfiguraci vnitřní paměť FLASH s udávanou výdrží 10.000 cyklů mazání / zápis. Xilinx už tyto obvody nevyrábí, přesto jsou k sehnání a pro amatérské konstrukce jsou stále vhodné, protože na rozdíl od mnoha pozdějších obvodů dokáží pracovat s pětivoltovou logikou.

Obvody CPLD už stěží kdokoli naprogramuje ručně. K vývoji se používají HDL (Hardware Definition Language), typicky VHDL nebo Verilog. V těchto jazyčích (později se důkladně seznámíme s VHDL) popisujete hardware pomocí výrazů, které definují buď propojení menších celků, nebo jejich chování. Vývojářské nástroje převedou takto zapsané výrazy do velkého souboru dat pro konkrétní obvod, a pomocí programátoru (většinou typu JTAG) se data zapíšou do obvodu CPLD.

## **FPGA**

---

Dostali jsme se k těm nejvýkonnějším programovatelným obvodům. Dokáží nahradit desítky tisíc až desítky milionů logických hradel a nejnovější obvody tohoto typu jsou svou složitostí a strukturou srovnatelné se současnými mikroprocesory.

Obvody FPGA (Field-Programmable Gate Array) rozšiřují koncept CPLD a posouvají jej opět o řád dál. Kromě makrobuněk a kombinační logiky, která ve FPGA bývá řešena pomocí tabulek (LUT, Look-Up Tables), nabízejí tyto obvody další funkční celky, jako jsou PLL pro generování frekvencí, paměti, násobičky, AD a DA převodníky, ...

Na rozdíl od obvodů CPLD mívají obvody FPGA svou konfiguraci uloženou nikoli ve vnitřní paměti, ale v paměti vnější, nejčastěji v podobě sériové FLASH. Po startu systému se z této paměti načte konfigurace do FPGA.

## **K čemu mi je FPGA?**

---

Přesně! To je vážná průmyslová věc, to není hračka pro bastliče! Jak bych ty řeči slyšel. Možná je slýcháte taky, a bojíte se světem FPGA vůbec zabývat, protože *na to přeci musíte být dírkovaná inženýrka*, abyste tomu rozuměli.

Mám dobrou zprávu: Nemusíte být dírkovaní a můžete si FPGA skvěle užít. Samozřejmě být vámi bych se nepouštěl do návrhu průmyslových obvodů, tam je potřeba přeci jen kromě zkušeností i nějaký teoretický základ, ale na takové to domácí hraní je FPGA docela fajn. Představte si to: V jednom takovém obvodu si můžete vytvořit osmibitový procesor, paměť, sériový UART, displej s výstupem na televizi nebo VGA a rozhraní pro klávesnici a SD kartu. Celý počítač. V jednom obvodu. Za pětikilo! No není to sen?

Ale než se do toho pustíme, tak mi dovolte kus nezbytné teorie, tentokrát formou otázek a odpovědí.

- **Jaké FPGA?**

Pro amatérské použití se nehodí nejnovější a nejvýkonnější obvody. Jejich schopnosti jsou daleko před potřebami amatérské praxe a jejich cena vysoce nad možnostmi amatérské penězenky. Ale i v té spodní, dostupné části spektra nalezneme dostatek obvodů pro konstrukci velmi zajímavých zařízení. Cílem těchto stránek je představit si základní kity, které pořídíte

za ceny do tisíce korun, což je rozumná částka, kterou domácí rozpočet unese. Ta nejjednodušší kombinace, viz dál, vyjde cca na 500 Kč.

- **Kdo vyrábí FPGA?**

Největší dva výrobci jsou Xilinx a Altera. Kromě nich vyrábí FPGA i další firmy, např. Lattice.

- **V čem se píše pro FPGA?**

FPGA jsou programovatelná logická pole, je tedy třeba je naprogramovat. Nejznámější jazyky jsou VHDL a Verilog, ale používají se i jiné (SystemC např.)

- **Xilinx, nebo Altera / Intel?**

To jsou dva největší výrobci FPGA. (Alteru před nedávnem koupil Intel.)

Jejich řady jsou do určité míry srovnatelné, ale navzájem nekompatibilní. Od Xilinxu pravděpodobně využijete řadu Spartan, konkrétně obvody z řad Spartan 3 a Spartan 6. Od Altery pak řadu Cyclone, konkrétně Cyclone II a Cyclone IV.

Platí, že vyšší řada nabízí větší obvody s více logickými celky, do kterých se vejde větší a složitější konstrukce. Volba výrobce ovlivní i další rozhodování. Podle výrobce použijete vývojové prostředí (ISE WebPack nebo Quartus II), a každý výrobce používá jiný JTAG. Jazyky naštěstí můžete použít u obou stejně.

Neexistuje obecná rada, jestli Xilinx nebo Altera. Já dlouho upřednostňoval Xilinx, teď mi připadají kity s Alterou dostupnější a propracovanější.

Styl práce se zas tak moc neliší. Pokud s FPGA začínáte, zvolte si jednu z těchto možností, později to můžete změnit. A jestli nevíte jakou, vyberte Alteru – důvod je, že na eBay koupíte levné čínské programátory pro Alteru levněji než levné čínské programátory pro Xilinx. *Navíc se mi zdá, že Quartus od Altery překládá VHDL rychleji než Xilinx ISE.*

Stručně: **Pokud jste začátečník a nevíte, kterého výrobce zvolit, odpověď zní Altera.**

- **VHDL, nebo Verilog?**

Další rozhodování se bude týkat použitého jazyka. VHDL i Verilog jsou použitelné jazyky pro všechny FPGA i CPLD, je tedy na vás, co si vyberete.

Oba jazyky jsou si do určité míry podobné svými schopnostmi a přístupem. Pro začátek si ale vyberte jeden, a ten se naučte. **Pokud nevíte jaký, bude to VHDL.**

VHDL je populárnější v Evropě, v USA spíš Verilog. VHDL je trošku víc přístupný lidem, kteří mají zkušenosť s programováním. Jinak je to oblíbené dilema, o kterém se lze mnoho hodin přít. (Pro klid duše: Ano, lze použít komponentu, napsanou ve Verilogu, ve vlastním projektu v VHDL, a je to snadné.)

- **Co budu potřebovat?**

### 1. Kit

Doporučuji pro úplný začátek malý kit s obvodem z rodiny Cyclone II: [EP2C5T](#). Malý, a přesto dostatečně výkonný, abyste v něm rozběhli např. osmibitový počítač s procesorem Z80, pamětí a BASICem. Pro zkušenější nebo náročnější mohu doporučit [velmi slušně vybavený kit s EP4CE6E22](#), kde najdete i SDRAM, FLASH, VGA nebo PS/2, a přitom ho lze stále koupit za velmi zajímavé ceny.

### 2. Programátor

Čínská kopie [USB Blasteru](#) funguje a je k sehnání doslova za pár korun

### 3. Vývojové prostředí (IDE)

Altera nabízí [Quartus II. Stahujte verzi 13.0 SP 1](#), ta podporuje použitý FPGA Cyclone II (novější jej už nepodporují). Ve verzi „Web Edition“ je zdarma.

### 4. VHDL

Najdete hned v další kapitole.

---

### *Jaký kit vybrat?*

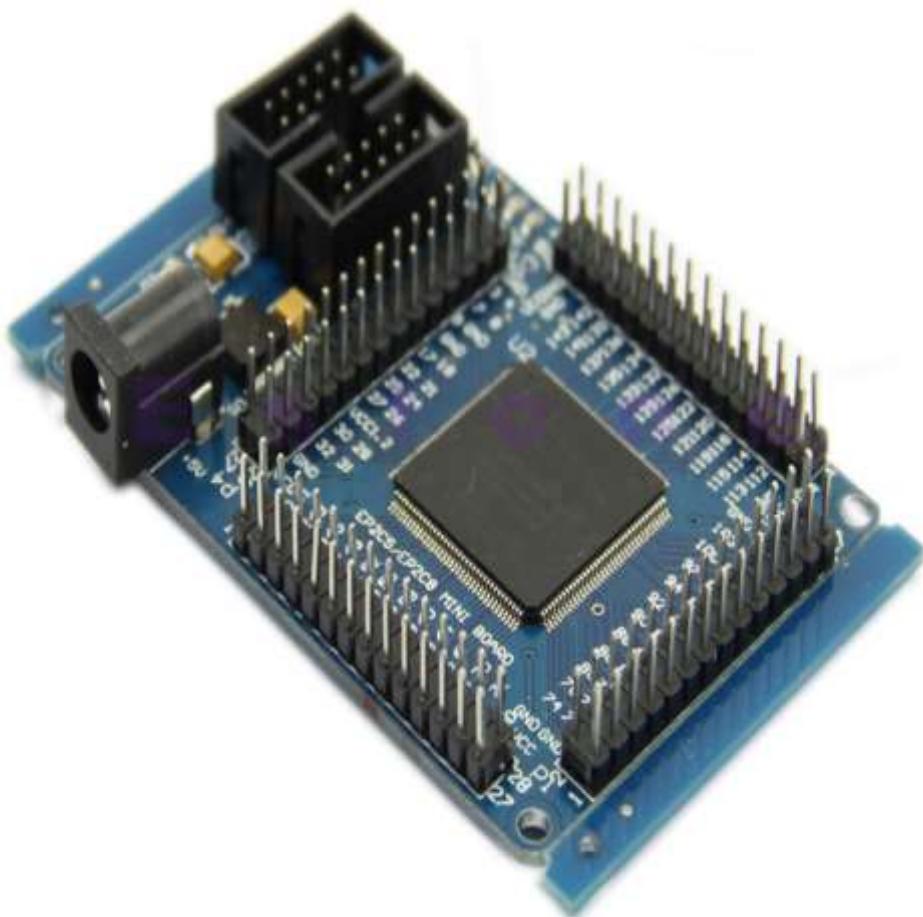
Já mohu doporučit velmi jednoduchý, levný, a přesto výkonný kit s obvodem Cyclone II od Altery, konkrétně EP2C5T144. Najdete ho na AliExpressu nebo na eBay. Klíčová slova jsou „EP2C5T144 FPGA Mini Development Board USB Blaster Programmer“ – za sadu včetně programátoru USB Blaster, respektive jeho čínské kopie, dáte okolo čtyř stokorun (leden 2019).

<https://8bt.cz/c2ebay/>

<https://8bt.cz/c2ali/>

---

Velká výhoda tohoto kitu je, že obsahuje vše nezbytné, ale nic navíc, takže máte k dispozici kompletní sadu vývodů. Nezapomeňte, že FPGA si většinou s pětivoltovou logikou moc nerozumí. Konkrétně tento obvod bude fungovat s logikou 3.3 V a nižší. 5 V na vstupu jej pravděpodobně zničí.



*První kroky s FPGA*

---

Máme kit, programátor, nainstalované IDE, připojili jsme to k PC přes USB, a co dál?

Nejprve troška teorie.

FPGA je po zapnutí úplně tuhý kus křemíku, který k tomu, aby něco zajímavého dělal, potřebuje nejprve nakrmit konfiguračními daty. Ty bývají nejčastěji uložené mimo FPGA, v sériové paměti FLASH, ale můžete je při ladění nacpat do FPGA i přes programovací rozhraní JTAG.

Kit EP2C5T144 například nabízí dvě rozhraní, do kterých lze zapojit USB Blaster: JTAG a AS. Pomocí JTAG dostanete konfiguraci do FPGA při ladění. Přeložit, nahrát, testovat... Po vypnutí a zapnutí se ale načte nová konfigurace zase z FLASH. Pokud chcete uložit konfiguraci přímo do této paměti, použijete AS (Active Serial). Existují i způsoby, jak nahrát obsah do FLASH přes JTAG, tzv. „indirect programming“.

Pro experimenty využijeme JTAG.

Vlastní návrh proběhne ve vývojovém prostředí – IDE. Ovládání IDE není triviální, ale pokud máte nějaké zkušenosti s vývojovým prostředím typu Visual Studio, Eclipse apod., brzy se sžijete i s těmito.

Obě se od sebe liší, každé má jinak pojmenovaná menu, ale základ je stejný: ke každému projektu existuje Project. Project v jednom adresáři schraňuje všechny soubory, potřebné k naprogramování FPGA. Jde především o popis funkce v některém z jazyků (VHDL, Verilog a další). K tému zdrojovým souborům si IDE vytvoří velké množství různých konfiguračních souborů (většinou je nemusíte editovat přímo, ale jsou na to v IDE nástroje). Jedním z takových nástrojů je nástroj, kterým můžete určit, který vývod FPGA má mít jakou funkci. V IDE Quartus se tato funkce jmenuje Pin Planner.

Překlad probíhá v několika krocích. Zase – názvosloví se liší, ale postup zhruba odpovídá.

- První krok je analýza a syntéza. V něm se ze zdrojových kódů vytvoří návrh pro konkrétní FPGA. Překladač zkонтroluje syntaxi, vyhodnotí logické výrazy, vazby mezi nimi, spočítá, kolik elementů bude potřeba, vytvoří seznam signálů, které bude potřeba připojit na piny FPGA...
- Ve druhém kroku se IDE snaží vhodně rozmístit komponenty do vnitřku FPGA. Zde se zohlední například i ten Pin Planner. Po

tomto kroku je jasno, zda se váš návrh do FPGA vejde, nebo zda je potřeba něco někde změnit.

- Třetí krok je vlastní překlad. Z výstupu druhého kroku se připraví konfigurační soubory, které se budou nahrávat do FPGA.
- Následují nejrůznější testy, hledání kritických míst, a závěrečný report, z něhož se např. dozvíte, nakolik jste využili možností svého FPGA.

Zde překlad končí. Další krok je vlastní programování do FPGA.

Kroky jsou přehledně vidět v IDE, takže víte, co už je splněno a co vás ještě čeká.

Nemusíte psát nutně všechno ve zdrojovém kódu – IDE obsahují i nástroje pro vizuální návrh, kde si obvod sestavíte, jako byste ho kreslili v Eagle nebo jiném EDA nástroji.

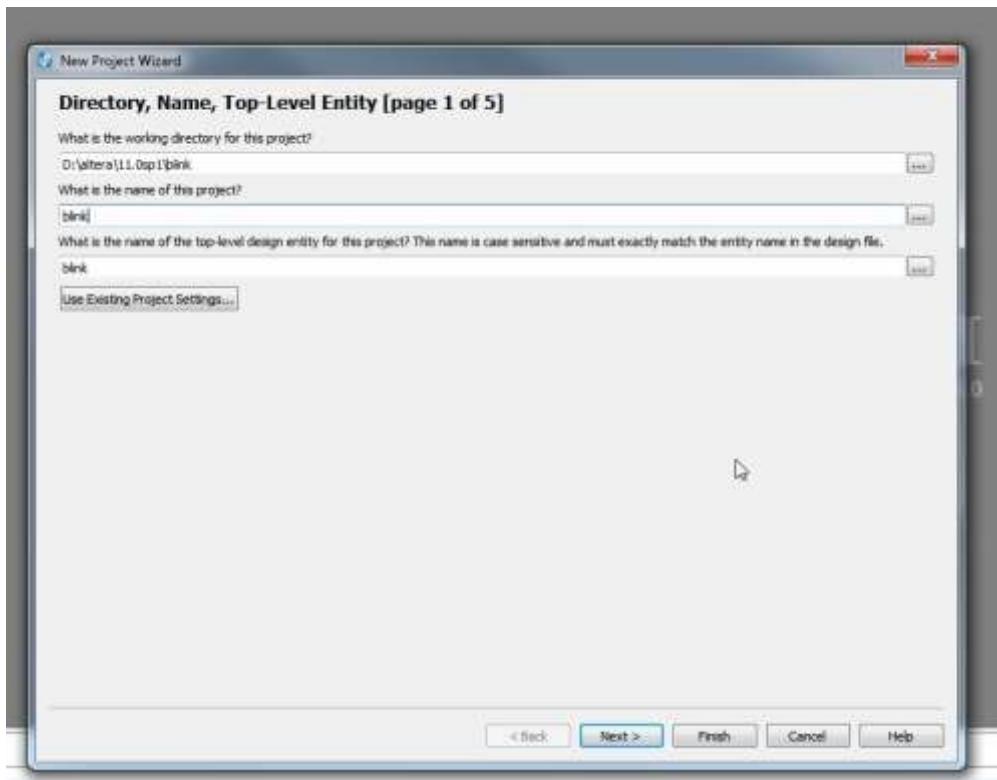
### **Hello world, model FPGA**

Pokud máte všechno potřebné, tj. kit, programátor i IDE, můžete si zkusit „blikat LEDkou“, což je taková hardwarová obdoba Hello world.

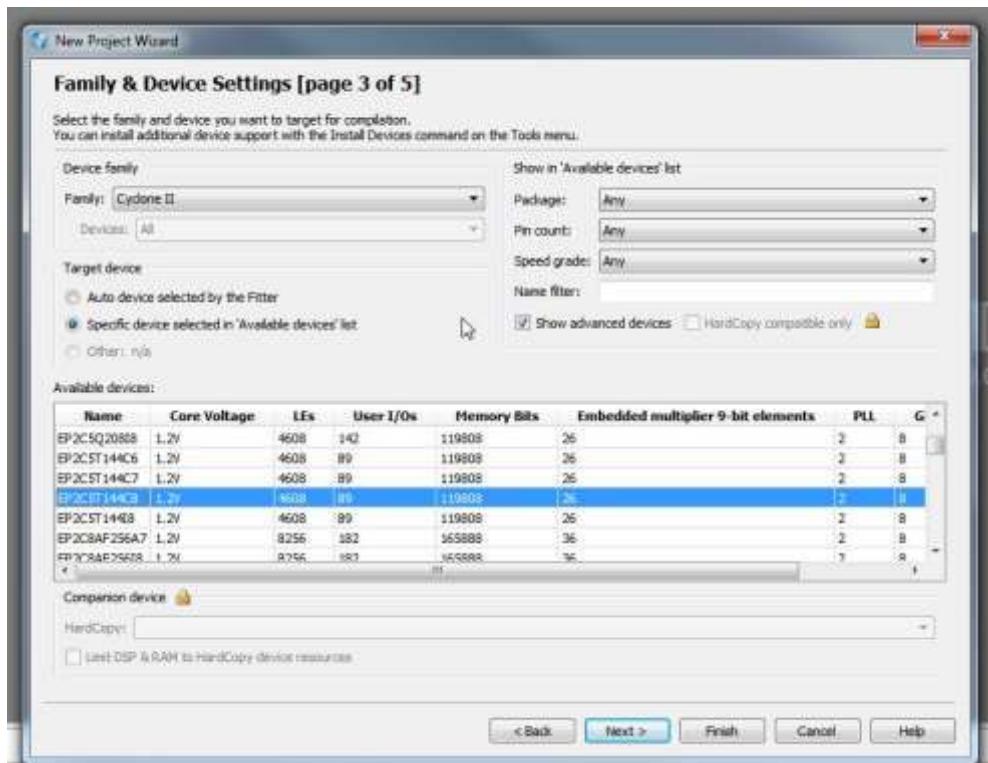
Budu popisovat blikání LEDkou pro [kit EP2C5, programátor USB Blaster](#) a [IDE Quartus](#).

#### **1. Vytvoření projektu**

Začínáme vytvořením projektu. Jako vždy: File – New Project Wizard.



Projekt pojmenujeme „blink“. Klikáme na NEXT.



Ve třetím kroku je potřeba vybrat použité FPGA. Zadejte rodinu „Cyclone 2“, čip je „EP2C5T144C8“.

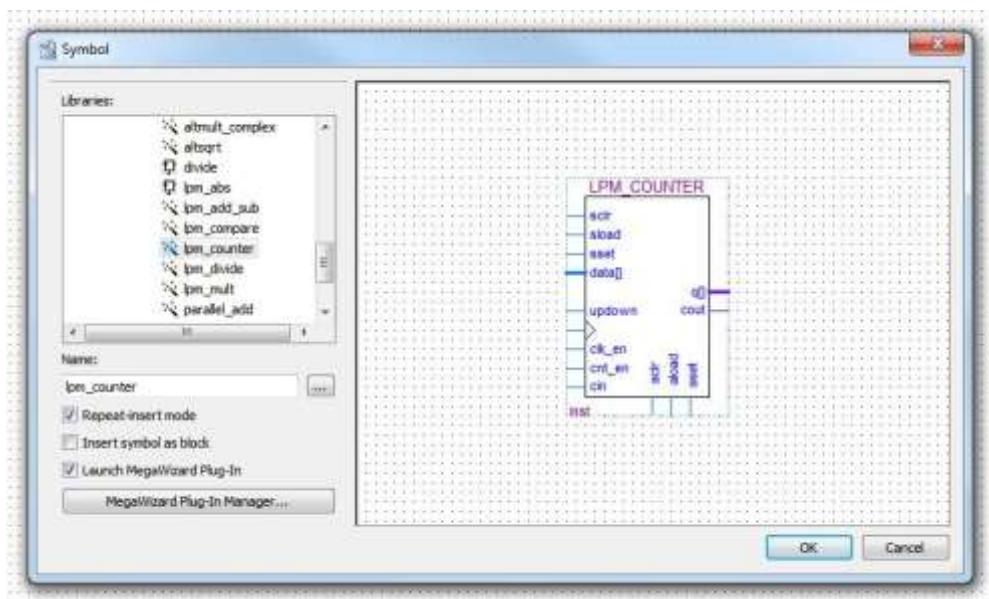
Next, next, finish...

## 2. Zapojení

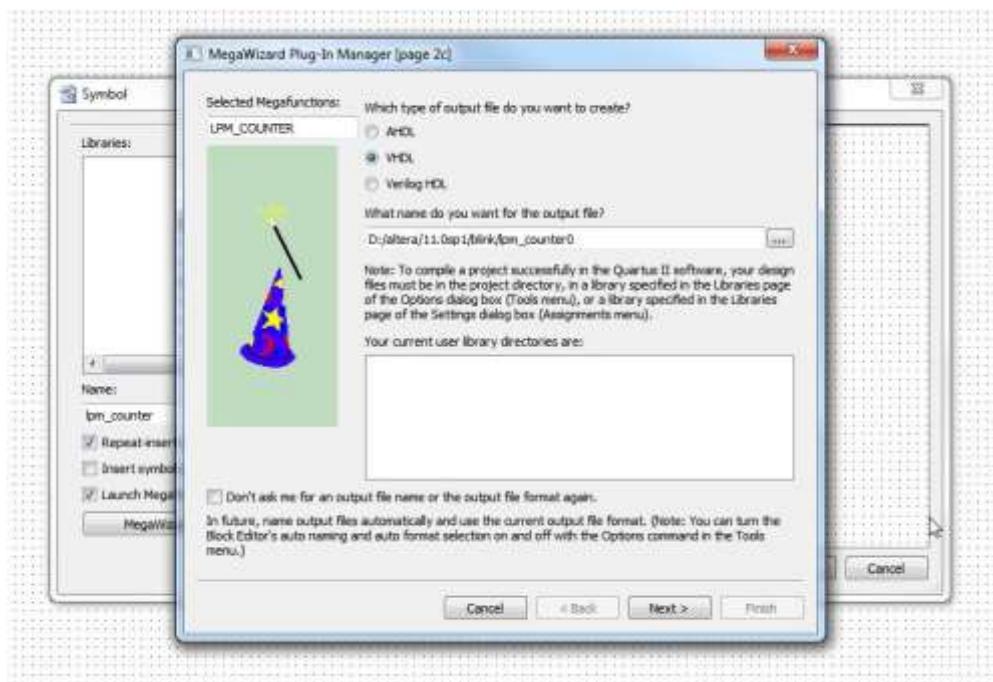
Blikání můžeme udělat mnoha způsoby. Já vybral ten, kdy si na hodinový vstup (50 MHz) připojíme čítač, kterým podělíme frekvenci natolik, aby bylo blikání pozorovatelné pouhým okem. To znamená ideálně 24 bitů a víc. Kit má navíc 3 LED, takže nechám blikat všechny tři a zapojím je na bity 24, 25 a 26. Dělič nemusím vytvářet z elementárních obvodů – Quartus obsahuje knihovnu neskomorně nazvanou Megalibrary, která obsahuje sadu nejrůznějších obvodů, od jednoduché logiky až po komplexní obvody typu řadiče SDRAM, rozhraní PCIe nebo síťové vrstvy PHY. Je mezi nimi i univerzální čítač.

Jak jsem slíbil výš, nebudeme obvod popisovat zdrojovými kódy, ale na-kreslíme si ho. Vyberu tedy File – New – Block Diagram/Schematic File.

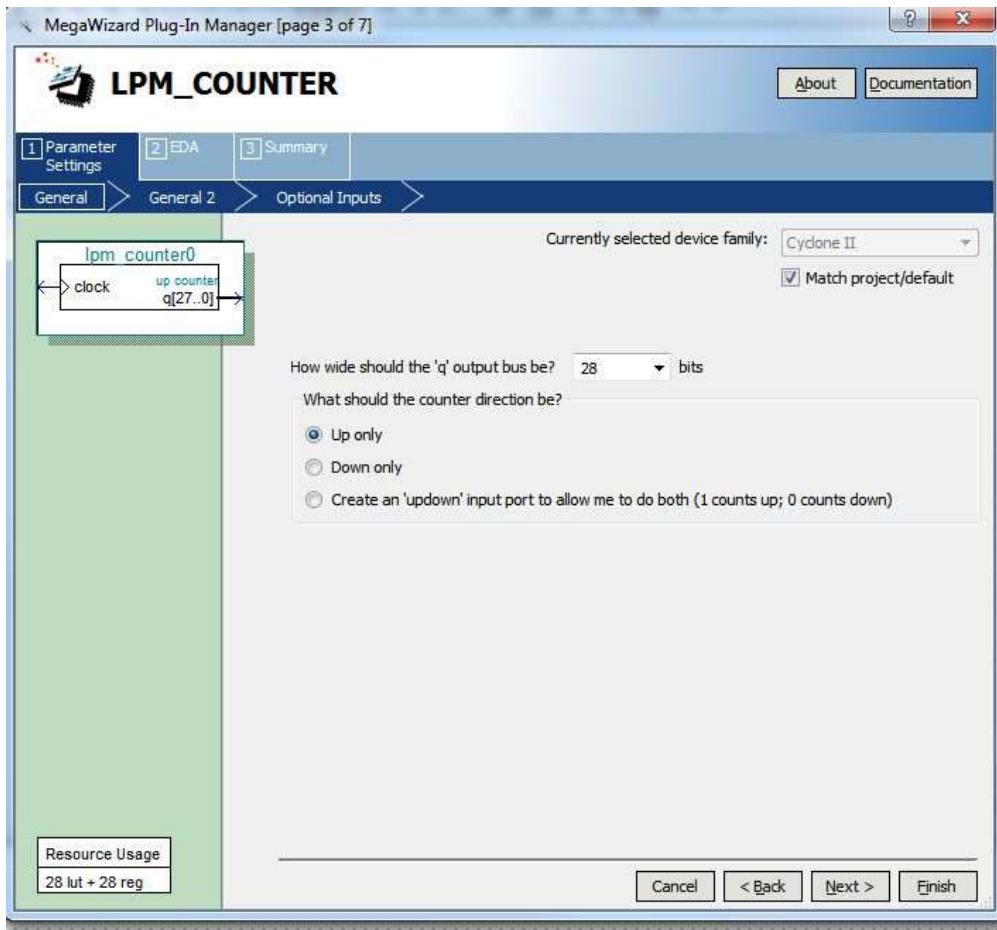
Otevře se známý „tečkovaný papír“, kam můžu umístit komponenty a propojovat je.



Nejprve tedy umístím komponentu. Vyberu si z „Megafuctions“, složky „Arithmetic“ obvod, který se jmenuje „lpm\_counter“. Kliknu na OK, a otevře se průvodce nastavením.



Jako jazyk zvolím VHDL (ted' je to jedno) a pojmenování nechám takové, jaké mi průvodce nabízí.

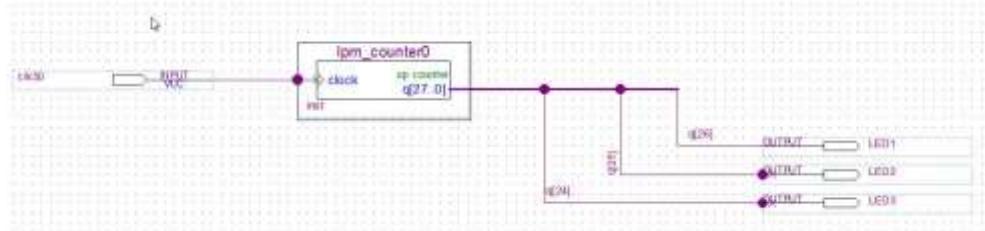


V dalším kroku zvolím bitovou šířku. Já zvolil 28 bitů a čítání nahoru (mohu vytvořit i čítač dolů, popřípadě obousměrný). Další volby nechám tak, jak jsou nastavené, na konci kliknu na Finish.



Objeví se okno, které mě upozorňuje, že jsem stvořil komponentu, a ptá se, jestli ji chci vložit do projektu. Chci, takže Yes.

Ted\x9e kresl\xedm zapojení. Na vstup p\x9eipoj\xedm vstupní pin, pojmenuju ho clk50. Na výstup p\x9eipoj\xedm sběrnici (Bus), která se bude jmenovat „q[27..0]“ – tedy 28 linek. Vyu\x9eiju z nich ale jen tři. Připrav\xedm si tři výstupní piny LED1-3 a p\x9eipoj\xedm je ke sběrnici jako q[24], q[25] a q[26].

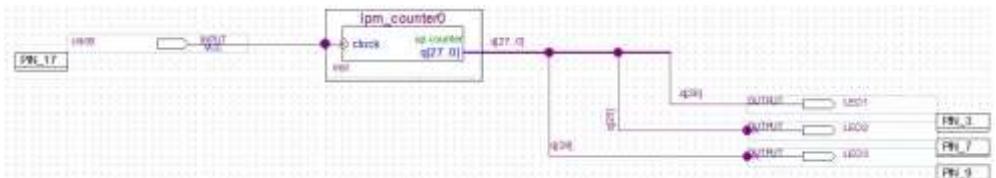


Soubor ulo\x9e\xedm jako „blink.bdf“, zkus\xedm si zadat p\x9eeklad (Processing – Start – Analysis & Synthesis, *Ctrl-K*), a pokud je vše OK, je na\x9ease p\x9eipoj\xedt piny z nákresu k fyzickým. Otev\xedu si Pin Planner (Assignments – Pin Planner) a zadám správné piny.

Podle [schématu](#) je hodinový vstup 50 MHz p\x9eipojen\xed na pin 17 a LED jsou p\x9eipojen\xed k pinům 3, 7 a 9. Stejn\xe9 tedy p\x9e\xedřad\xedm i piny v planneru.

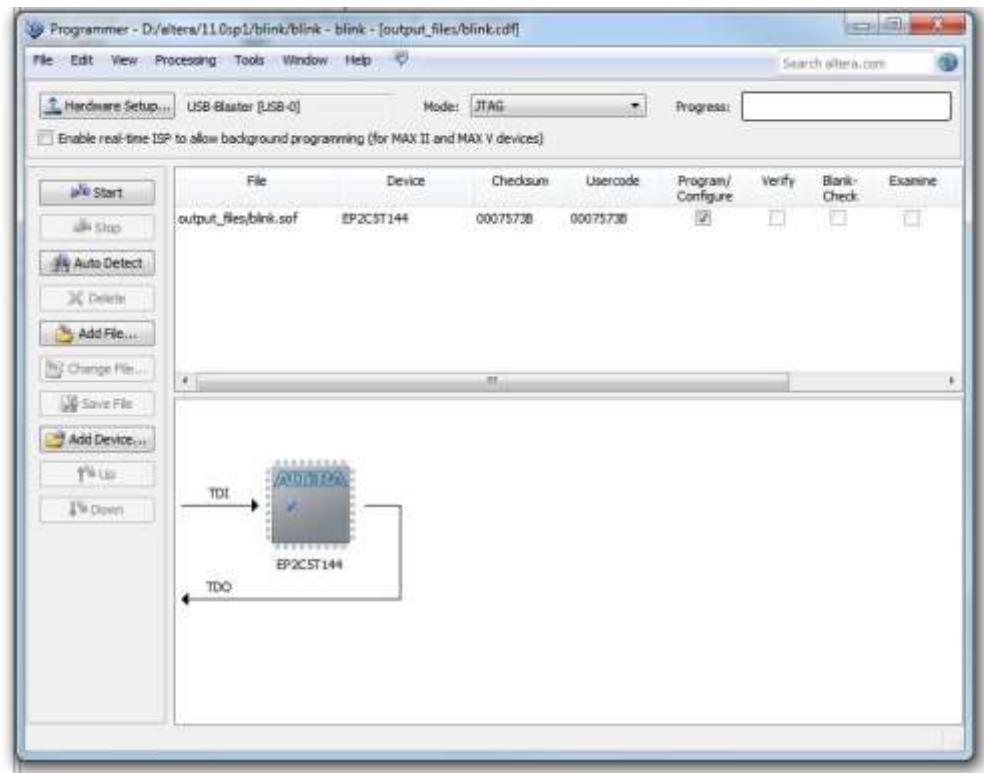
Node Name	Direction	Location	I/O Bank	I/O Group	I/O Standard	Reserved	Current Strength	Differential Pair
cK50	Input	PIN_17	3	B1_N0	3.3-V_LV_default		2mA (default)	
LED1	Output	PIN_2	1	B1_N0	3.3-V_LV_default		2mA (default)	
LED2	Output	PIN_3	1	B1_N0	3.3-V_LV_default		2mA (default)	
LED3	Output	PIN_9	1	B1_N0	3.3-V_LV_default		2mA (default)	

Po zavření planneru vidím, že se přiřazení promítlo do schématu:



Teď tedy mohu spustit celý překlad (Processing – Start Compilation, *Ctrl-L*). Měl by proběhnout bez chyb.

Pokud je vše v pořádku, je načase programovat. Připojte USB Blaster a počkejte, až se nainstaluje. Možná bude potřeba vyřešit ovladače... Blaster připojte do konektoru JTAG na kitu a kit zapojte na zdroj 5 V. Měla by svítit POWER LED. Otevřete programátor (Tools – Programmer), vyberte jako nástroj „USB Blaster“, mód „JTAG“ (pokud byste programovali natrvalo, viz výše, zde zadáte AS). A pak už stačí jen kliknout na Start.



Během několika sekund se LED na kitu rozblíkají. Hurá!

Po chvíli zjistíte, že blikají nějak divně, že to počítání moc neodpovídá, jako by snad počítaly směrem dolů, a při pohledu na zapojení kitu vám to dojde: LED nejsou připojené na zem, ale na Vcc. Tedy inverzně. Jako intelektuální cvičení si můžete zkousit, jak do celého zapojení přidat tři invertovery...

# VHDL

## Proč se učit VHDL?

Odpověď je jednoduchá: Pokud chcete používat FPGA, (skoro) nic jiného vám nezbývá.

Tedy samozřejmě, můžete místo VHDL zvolit Verilog, můžete se učit System C, můžete na tyhle jazyky rezignovat a všechno malovat jako schéma, ale garantuju vám, že se znalostí VHDL či Verilogu bude váš život s FPGA snazší.

Otázka „VHDL, nebo Verilog“ je další z mnoha nekonečných programátorských debat, kde není jednoznačná odpověď. Já jsem zvolil VHDL. Jak říká klasik: Zkusil jsem obojí, a VHDL mi přišlo lepší. VHDL se víc prosadilo v Evropě, Verilog v USA. Rozdílů mezi těmito jazyky je mnoho, především syntaktických, ale i principiálních. Dá se s jistou mírou nepřesnosti a zjednodušení říct, že Verilog se snaží přiblížit syntézu k programování, zatímco VHDL vám dává větší možnosti ovlivnit chování celku.

Když tedy máte jasno v tom, jaký jazyk zvolit, je potřeba se ho naučit. Hodně pomůže, když umíte v něčem programovat, ještě víc pomůže, když chápete princip elektronických zařízení, a úplně nejvíce pomůže, když jste si už něco navrhli, postavili a ono to fungovalo!

K učení nepotřebujete nezbytně nutně hardware. Dá se psát „nanečisto“ a v nějakém IDE (jsou i pro Linux, nebojte) si simulovat chování, ale rovnou říkám, že bude lepší si pořídit nějaký kit. Není to nic extrémně nákladného, a ty základní lze pořídit i s programátorem a poštovním někde okolo šesti stovek.

A pak už jen sedněte, proberte se všemožnými odkazy, dívejte se, co všechno se dá s FPGA udělat... to je ta nejlepší motivace se to začít učit taky!

## Než začneme...

- Quartus verze 13.0.1 Web Edition – v novějších není podpora pro čipy Cyclone II a Cyclone IV, které používáme
- Nějaký kit – viz sekce o FPGA, já budu používat doporučený kit s EP2C5

## Úplné základy a nezbytná teorie

Pokud k VHDL přistupujete se stejnými základy, jako jsem měl já, budete mít problém. Pojďme se podívat na nejčastější příčiny nepochopení, které u VHDL hrozí programátorům.

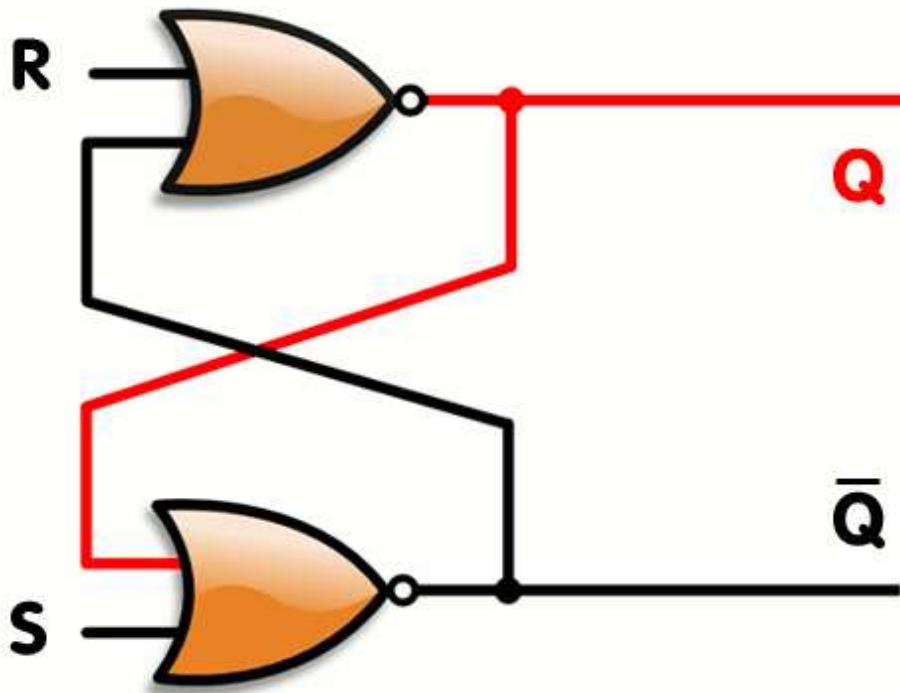
VHDL patří do rodiny jazyků HDL, což znamená „Hardware Description Language“. Návrháři už v názvu naznačují, že jde o jazyk popisný (description), nikoli programovací (programming) – zdůrazňuju to proto, že programátoři mají tendenci každý „jazyk“ považovat za programovací, a pokud v něm nejde zapsat algoritmus (HTML například), tak prohlásí, že nestojí za nic. V jazycích HDL se rovněž moc neprogramuje.

To „V“ znamená VHSIC, což je další akronym z Very High Speed Integrated Circuit, tedy „velmi rychlé integrované obvody“.

VHDL tedy popisuje nějaké vnitřní zapojení integrovaných obvodů (navíc vysokorychlostních, ale to nechme stranou). Není svázané s konkrétním obvodem ani technologií a lze jej použít k popisu digitálních zapojení. Na základě tohoto popisu pak specializované nástroje připraví podklady pro naprogramování CPLD, FPGA nebo třeba vlastního IO.

### Jak popsat IO?

Máme několik možností, jak popsat integrovaný obvod ve VHDL. Představme si takový klopný obvod R-S:



V první řadě si ho popíšeme jako „black box“, tedy jako krabičku, která má dva vstupy (R, S) a dva výstupy (Q, /Q).

Pro programátory: Tohle je něco jako deklarace. Je to něco, co se jmenuje „klopný obvod R-S“, a nabízí to pro komunikaci se světem tyto možnosti. Po implementaci nepátráme, ta je někde jinde.

V terminologii VHDL jsme právě popsali **entitu** pomocí jejího **portu**. Entita „RS“ má tento port: jednobitový vstup R, jednobitový vstup S, jednobitový výstup Q a jednobitový výstup /Q.

Popis zvenčí bychom měli, ovšem ten nám nic neříká o tom, co se děje uvnitř. Ve VHDL máme rovnou tři možnosti, jak popsat vnitřní strukturu.

### 1. Strukturní popis

Strukturní popis se zaměřuje na funkční celky a jejich propojení. V tomto případě tedy řekneme, že uvnitř jsou dvě hradla NOR, nazveme si je G1 a G2, každé hradlo NOR má zase deklarovaný port, řekněme A, B a Y, a zaměříme se na to, jak jsou propojena. Tedy G1 má svůj vstup A připojený na vstup R, vstup B na výstup /Q a svůj výstup Y na výstup Q... a tak dál. (Sku- tečnost bude o něco málo složitější...)

## 2. Data flow

Popis „data flow“, neboli toku dat, se nezaměřuje na jednotlivé komponenty, ale na to, jak se obvodem šíří data, respektive „kde se vezme výsledek?“ V tomto případě řekneme, že  $Q$  je výsledek funkce NOT ( $R$  or  $/Q$ ), a  $/Q$  je NOT ( $S$  or  $Q$ ). A máme to. (Ve skutečnosti to nemáme, protože takhle jednoduché to ve VHDL není, musíme použít jinou techniku, ale princip je tento.)

## 3. Behaviorální popis

Tento styl popisu se asi nejvíce blíží klasickému programování a lidi, kteří mají zkušenosť s programováním, budou mít tendenci vše řešit takto. Což je postup, před kterým varuji rovnou, a ještě tak asi dvacetkrát varovat budu... Behaviorální popis má své nezastupitelné místo a výrazně zjednoduší návrh, na druhou stranu ale není všespásný a nelze si myslet, že „prostě naprogramuju chování obvodu“. V takovém případě jste špatně a hledáte kurz programování v assembleru. Ale zpět k behaviorálnímu popisu. V našem případě bychom definovali **proces**, který se spustí, pokud dojde ke změně na vstupech  $R$  nebo  $S$ , a vyhodnotí jednoduchou funkci: Pokud  $R$  je 1 a  $S$  0, tak  $Q$  bude 0,  $/Q$  bude 1, jinak pokud  $R$  je 0 a  $S$  je 1, tak  $Q=1$ ,  $/Q=0$ , jinak pokud jsou oba v nule, tak  $Q$  i  $/Q$  zůstávají stejné jako předtím, no a pokud jsou oba v jedničce, tak *něco*, protože *správně to je nedefinovaný stav*. V procesu můžete použít právě podmínky, větvení, cykly a další programátorské výmožnosti, ovšem s výraznými omezeními.

V praxi se všechny tři přístupy kombinují podle toho, jak je to pro danou chvíli vhodné. Něco se líp zapíše pomocí propojení vývodů menších celků, něco zase pomocí toku dat, něco je nejlépe popsát procesem.

## Chytáky

Asi největší chyták, do kterého se může programátor lapit, je fakt, že „příkazy zapsané pod sebou“ neznamenají, že se provedou „po sobě“. Když si představíte logický obvod, tak tam není nějaké „před“ a „po“, tam se uvažuje s ideálně nulovým zpožděním, takže změna vstupu se okamžitě projeví v celém systému naráz. (Ve skutečnosti ne, protože každý člen má nějaké svoje zpoždění, a při vysokých rychlostech je s ním potřeba počítat, ale pro tuto chvíli zpoždění zanedbejme.) Představa, že *nejdřív něco změ-*

*ním, pak se něco provede, pak zase změním něco jinak* je ve světě logic-kých obvodů mylná. Ano, sekvenční operace lze udělat, ale musíte si pro ně nejdřív vytvořit *stavový automat*. Představte si popis architektury (data flow, behaviorální) nikoli jako posloupnost příkazů, které jdou po sobě, ale jako seznam operací, které se provádějí najednou a konkurenčně. Od-povídá to realitě: v obvodech pracují všechny části současně a naráz, není tam nic, co by je postupně přepínalo mezi stavý.

Druhý chyták je syntax. Lehce, ale fakt jen velmi lehce, připomíná Pascal s jeho klíčovými slovy begin a end. Středník někde být musí, někde nesmí, elseif není ani „elseif“, ani „elif“, ani „else if“, ale „elsif“, hodnota pro-měnné se přiřazuje pomocí := (ale velmi podobné přiřazení signálu se dělá pomocí <=) a celé to je silně typované. Jinak je VHDL tolerantní vůči vel-kým / malým písmenům, nijak neřeší mezery ani odsazení a je v tom velmi tolerantní.

A ještě takový terminologický detail: Překlad, tedy to, co z programování známe jako komplikaci kódu, tu není komplikace, ale syntéza, a neprovádí ji komplikátor, ale syntetizér. On totiž nedělá to, že by kompiloval příkazy, on vytváří (syntetizuje) popsaný obvod...

## HELLO WORLD!

Lžu. Ještě ani zdaleka ne. Křivka učení je hodně povlovná, a ještě musíme pář věcí probrat, než si blikneme LEDkou...

V úvodu jsem psal, že VHDL je deklarační a popisný jazyk (nikoli imperativní) a že je na první pohled trochu blízký Pascalu. Pojďme si ukázat zá-kladní koncepty.

Naše stavební bloky ve VHDL jsou entity. Entita je popsána jednak svým rozhraním navenek (viz minule zmiňovaný **port**), jednak svou vnitřní architekturou. Ta může být popsána několika způsoby, opět viz úvod, a v praxi se nejčastěji potkáte s jejich mixem.

Pojďme si nejprve nadefinovat entitu, která bude provozovat neúplné jed-nobitové sčítání. Jak to funguje?

Při sčítání dvou jednobitových hodnot je pravidlo prosté:

- $0+0 = 0$
- $1+0 = 1$

- $1+1 = 10$  – a protože sčítáme jednobitově, tak je výsledek 0 a nastavený přenos.

Naše entita tedy bude mít dva vstupy, A a B (vstupní přenos neuvažujeme, proto *neúplná sčítačka*), a dva výstupy, Q a Cout. Můžeme si sepsat pravidostní tabulku...

A	B	Q	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Můžeme s tím ještě laborovat dál, ale u téhle jednoduché funkce na první pohled vidíme, že Q je  $A \oplus B$ , Cout je  $A \text{ AND } B$ . Můžeme zvolit strukturní zápis (tedy *jak je to zapojeno*), ale tady bude vhodnější zápis stylem data flow (tedy *jak tečou data*).

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

-- neúplná sčítačka

entity adder is
    port (
        A, B: in std_logic;
        Q, Cout: out std_logic
    );
end entity adder;

architecture main of adder is
begin
    Q      <= A xor B;
    Cout <= A and B;
end architecture;
```

Rozebereme si to po jednotlivých blocích.

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

Tyto dva řádky se snad raději naučte nazepaměť jako říkadlo. Znamenají, že budeme používat standardní knihovnu, definovanou organizací IEEE, a z této knihovny využijeme tu část, kde jsou definované standardní pojmy, související s logickými výrazy – jednobitové logické hodnoty, vícebitové vektory apod.

-- neúplná sčítáčka

Poznámky začínají dvěma znaky „minus“. Cokoli od nich dál až do konce řádku je poznámka.

Raději hned teď upozorním na jednu věc, která je schopna nadělat spoustu zlé krve: Základním logickým typem je std\_ulogic, který používá devítihodnotovou logiku. Definované hodnoty jsou:

Označení	Hodnota
,U'	Neinicializováno (uninitialized)
,X'	Nedefinovaná hodnota mezi 0 a 1
,0‘	Logická 0 (silná)
,1‘	Logická 1 (silná)
,Z‘	Vysoká impedance (3. stav v třístavové logice)
,W‘	Nejistá hodnota mezi L a H (slabě buzená)
,L‘	Slabá logická 0
,H‘	Slabá logická 1

Označení	Hodnota
,	Hodnota, která nás nezajímá

Důvod, proč jsou zavedeny všechny ty nejrůznější slabé hodnoty, je ten, aby bylo možné vytvářet různé „montážní OR“ a „montážní AND“ a aby bylo snazší emulování obvodů. V praxi *byste měli* používat právě std\_logic. Klasická „dvouhodnotová“ (ve skutečnosti jich má více) logika je z ní odvozená a jmenuje se std\_logic. Její použití může být nevhodné, protože v některých situacích musí syntetizér (to je ten software, který převádí VHDL na konkrétní fyzickou reprezentaci) používat resolve funkci, která jasně rozhodne, jestli je signál 0, nebo 1, což může návrh zesložitit nebo zpomalit syntézu. Na druhou stranu, když si necháte nějakou komponentu vygenerovat, nebo použijete hotový návrh, bude používat s největší pravděpodobností právě std\_logic. Já budu v příkladech používat std\_logic.

Pokračujme dál.

```
entity adder is
  port (
    A, B: in std_logic;
    Q, Cout: out std_logic
  );
end entity adder;
```

Slovo „entity“ uvozuje deklaraci, tedy tu část, kde popíšeme rozhraní. Tvar, jaký je uvedený výše, je ten nejčastější, s jakým se setkáte. Obecně:

```
entity {jméno} is
  port (
    {signál} [, {signál2}...]: {mód} {typ} [; ...]
  );
end [entity] {jméno};
```

Jméno entity je zcela na vás, ale musí dodržovat základní pravidla pro pojmenování, podobná těm v ostatních jazycích:

- Obsahuje velká a malá písmena, číslice a podržítko
- Začíná písmenem (ne číslicí ani podržítkem)
- Nerozlišují se velká a malá písmena (ADDER je totéž co Adder)
- Nesmí končit podržítkem

- Nesmí obsahovat dvě podtržítka za sebou (Takhle \_\_ Ne)

Deklarace je ukončena slovem „end“. Ve VHDL je mnoho ENDů (end if, end component atd.), a proto je dobré zvyknout si psát, k jaké struktuře se ten který end váže. Zde třeba „end entity“. Jak entity, tak její jméno je u „end“ nepovinné, ale může být. Radím zvyknout si, že vždy píšete, k čemu end patří.

Uvnitř deklarace je pouze *port()*. Ještě se zde může vyskytnout část generic, ale o té později, popř. část definic, deklarací a příkazů. V sekci *port()* – závorky zde musí být – je deklarováno, jaké má daná komponenta rozhraní. Seznam jednotlivých signálů se skládá z položek ve tvaru „*jméno signálu: mód typ*“, oddělených středníkem. Za poslední deklarací signálu nesmí být středník! Naopak musí být za závorkou, ukončující *port()*! Mnoho chyb takhle vzniká...

Jméno signálu má zase stejná pravidla jako jiná jména, viz výše, mód je IN, OUT nebo INOUT, která naznačují, jestli signál do obvodu vstupuje, vystupuje z něj, nebo jestli je obousměrný.

Typ je například už výše zmiňovaný std\_logic. Může to být i std\_ulogic, popřípadě vektor (tedy několik signálů spojených do jednoho vícebitového), anebo něco zcela jiného (integer, uživatelsky definovaný typ...) Pro tento chvíli zůstaňme u std\_logic.

Naše sčítačka má tedy dva vstupní a dva výstupní signály.

```
architecture main of adder is
begin
    Q      <= A xor B;
    Cout <= A and B;
end architecture;
```

V části „architecture“ je popsáno fungování obvodu (programátorským slangem jde o definici, zatímco entita byla deklarace). Tvar je obecně takový:

```
architecture {jméno architektury} of {jméno
entity} is
[ ... nějaké deklarace v rámci architektury
...
]
begin
```

```
[ ... příkazy ... ]
end [architecture] [{jméno architektury}] ;
```

Jméno architektury je zase libovolné. V naprosté většině případů budete vytvářet pro entitu jen jednu jedinou architekturu, a pak je úplně jedno, jak se jmenuje. Ale vězte, že architektur můžete mít pro jednu entitu více, každou jinak pojmenovanou, a v určitých případech (např. při testování) se na ně odvolávat. Architektura se vztahuje k nějaké entitě, a její jméno je zase uvedené v hlavičce

Za hlavičkou („architecture .... of ... is“) je část lokálních deklarací. Zde si definujeme typy nebo signály, které jsou použité v rámci architektury (analogicky: lokální proměnné v rámci bloku). Pak následuje vlastní výkonná část architektury, uvozená slovem „begin“, a celé to končí zase slovem „end“ – a stejně jako výš i tady doporučuju naučit se psát „end architecture“.

V naší sčítačce nepotřebujeme žádné lokální signály, jsou to vlastně jen dvě logické funkce. Bude nejjednodušší je popsat právě „data flow“ modelem, kdy řekneme, že „signál Q nabýde hodnoty A xor B“ a „signál Cout nabýde hodnoty A and B“.

Znovu opakuju: Není to tak, že by se NEJDŘÍV přiřadila nějaká hodnota do Q, a POTOM jiná do Cout. Obojí se provádí najednou, protože to syntetizér převede do zapojení logických obvodů. Není to program, je to popis toho, jak vznikají výstupní hodnoty. Pokud máte tendenci dívat se na tento zápis jako na zápis programu, považujte ho za „atomickou operaci“, která proběhne „najednou a nedělitelně“ a na konci bude mít nějaký výsledek.

## Testování

Nastal čas otestovat sčítačku z předchozí kapitoly. Nejprve jsem si vytvořil v Quartu (pardon, ale klasické vzdělání mi brání psát „Quartusu“) projekt.

Je potřeba dbát na to, že „hlavní entita“ se musí jmenovat stejně jako projekt, takže jsem projekt pojmenoval „adder“.

Místo popisovaného nakreslení obvodu ve vizuálním nástroji zvolím vytvoření VHDL souboru. File – New – VHDL file. Uložím si jej jako „adder.vhd“ („.vhd“ je standardní přípona VHDL souborů, můžete použít i „.vhdl“. Verilog používá „.v“).

Po spuštění překladu (Processing – Start – Analysis and Synthesis, též *Ctrl-K*) proběhne syntaktická kontrola a překlad. Pokud bylo něco špatně, Quartus zahlásí chyby, pokud bylo všechno OK, můžeme jásat.

Opravdu? No, ne tak docela. V programování je dobrým zvykem testovat, v elektronice taky. Jak se testuje ve VHDL? Princip je podobný.

Vytvoříme si testovací entitu (konvencí je pojmenovávat ji „test“ nebo „testbench“), ve které použijeme náš vytvořený obvod. Pomocí speciálního zápisu signálů (s určeným časem změny) připravíme pro testovaný obvod nějaké vstupní podmínky, a budeme se dívat, co se děje na výstupech. Uložím si ji do nového souboru (rozdělovat entity do souborů je taky dobrý zvyk) s názvem „testbench.vhd“.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test is
end;

architecture bench of test is

component adder
    port (
        A, B:  in std_logic;
        Q, Cout:  out std_logic
    );
end component;

signal tA,tB,tQ,tCout: STD_LOGIC;

begin

    tA <=      '0',
              '1' after 30 NS,
              '0' after 60 NS,
              '1' after 90 NS;

    tB <=      '0',
              '1' after 60 NS;

UUT: adder port map (tA,tB,tQ,tCout);

end bench;
```

Na začátku zase vidíme oblíbené deklarace použitých knihoven (dobře rádím: Naučte se nazepaměť!). Entita se jmenuje „test“ a je prázdná – nenašíz žádné rozhraní, žádný port navenek. To je v pořádku. Představme si testovací zapojení jako desku s testovací elektronikou, do které se zasouvá testovací součástka – taky nemá navenek žádné rozhraní.

Architektura popisuje zapojení našeho testeru. Všimněte si, že mezi řádkem „architecture bench of test is...“ a vlastním „begin“ jsou uvedené dvě deklarace. První je *deklarace komponenty*. Podobně jako v C máte v hlavičkovém souboru „prototyp funkce“, tedy jeho deklaraci s uvedenými typy vstupních proměnných a výsledku funkce, tak i ve VHDL se použitá komponenta musí nejprve nadeklarovat, aby překladač věděl, jaké jsou k dispozici porty.

Entita je tedy „deklarace“ nějaké součástky, architektura je její „definice“, a komponenta je deklarace při použití. Všimněte si, že část od „component adder“ po „end component;“ je doslova shodná s entitou adder z minulého článku, pro jistotu zkopíruju:

```
entity adder is
  port (
    A, B: in std_logic;
    Q, Cout: out std_logic
  );
end entity adder;
```

Jediný rozdíl je v tom, že slovo „entity“ je nahrazeno slovem „component“. Touto deklarací překladač ví, že je někde nějaká externí entita „adder“, kterou použije jako komponentu pro sestavování aktuálního obvodu.

Pod komponentou (může jich zde být samozřejmě více) je definice **signálů**. Je podobná definici vstupů a výstupů u entity, ale neudává se zde směr (in, out...) Signál si představme jako „drát“, který je někde uvnitř obvodu a není vyveden ven. Taková představa pro tuto chvíli stačí, ve skutečnosti je to o něco složitější, ale k tomu se dostaneme.

Pak už začíná vlastní popis toho, co se v testovacím obvodu děje. Už jsme viděli přiřazení hodnoty signálu nebo výstupu pomocí operátoru \_<=. Zde je použita jiná forma přiřazení, kdy na pravé straně není zapsaná hodnota, ale průběh signálu – v případě signálu \_tA\ se začíná v log. 0, po 30 nanosekundách přejde do log. 1 (,1' after 30 NS\), po 60 ns (od počátku simulace, ne od předchozího kroku) se změní zase do log. 0 a tak dál.

Všimněte si jedné důležité věci: **Logické hodnoty se zapisují v apostrofech!** Ve VHDL se totiž rozlišují logické hodnoty ('1', '0' atd.) a čísla (1, 4, 255). Pokud se pokusíte přiřadit hodnotu nějakému signálu s typem std\_logic nebo std\_ulogic pomocí něčeho jako `Q <= 1`, překladač vám vynadá...

Jsou tedy definované hodnoty signálů tA a tB, a to pomocí průběhů v čase. Je dobré si uvědomit, že takový zápis má smysl pouze v simulacích, ve vlastním obvodu takové hokus pokusy neuděláte, resp. syntetizér vás upozorní, že použije první hodnotu a zbytek ignoruje.

Poslední část architektury je strukturní zápis použití komponenty. Obecný tvar je

pojmenování:název komponenty port map  
(připojení portů) ;

Pojmenování je název, který v rámci architektury ponese instance komponenty. Kdybychom chtěli použít dvě sčítáčky, použijeme dvě různá jména. Název komponenty je stejný jako v deklaraci „component“, následují klíčová slova **port map** a v závorce seznam, podle něhož se komponenta do obvodu připojí. Seznam má kolik položek, kolik má deklarace port(), a ve stejném pořadí, v jakém jsou uvedeny vývody komponenty, uvedeme signály, které se na daný vývod mají připojit. Více si ukážeme v dalším pokračování. V tuto chvíli platí, že na vývod A připojíme signál tA, na vývod B připojíme signál tB atd.

Pojmenování je „UUT“, což je opět konvence pro testování: „Unit Under Test“.

Signály tQ a tCout jsou připojeny na výstupy testované komponenty, ale nijak se s nimi dál nepracuje. To nám nevadí, protože my s nimi zde pracovat nechceme. My si na ně pouze připojíme „sondu“.

Nadešel čas testu... Spusťte si Modelsim (Tools – Run Simulation Tool – RTL simulation, od instalace by mělo být vše správně nastaveno, pokud nemáte nastaveno, musíte si nastavit, že budete používat Modelsim).

Assignments – Settings – EDA Tool Settings a zde vybrat v řádku „Simulation“ možnost „ModelSim – Altera“. Pokud bude později Quartus protestovat, že není zadána cesta, zvolte „13.0sp1\modelsim\_ase\win32aloem“.

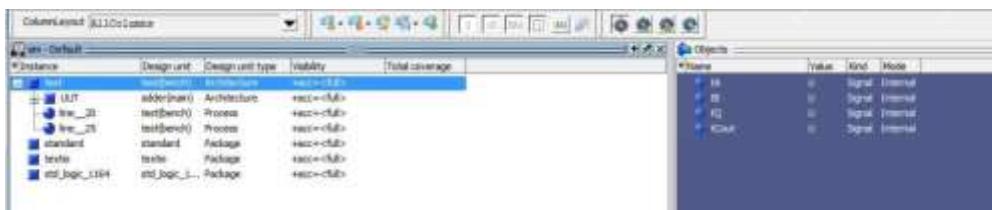
Rozhraní je mohutné, ale my ho zkoumat teď nebudeeme, soustředíme se jen na okno s knihovnou (Library), kde by jako první knihovna měla být uvedena knihovna „work“ – tedy ta, na které pracujeme.

Name	Type	Path
work	Library	rtl_work
+ E adder	Entity	D:/VHDL/webdemo/ch02/ch02.vhd
+ E ch02	Entity	D:/VHDL/webdemo/ch02/ch02.vhd
+ rtl_work	Library	D:/VHDL/webdemo/ch02/simulation/m...
+ 220model	Library	\$MODEL_TECH../altera/vhdl/220model
+ 220model_ver	Library	\$MODEL_TECH../altera/verilog/220m...
+ altera	Library	\$MODEL_TECH../altera/vhdl/altera
+ altera_lnsim	Library	\$MODEL_TECH../altera/vhdl/altera_l...
+ altera_lnsim_ver	Library	\$MODEL_TECH../altera/verilog/altera...
+ altera_mf	Library	\$MODEL_TECH../altera/vhdl/altera_mf
+ altera_mf_ver	Library	\$MODEL_TECH../altera/verilog/altera...
+ altera_ver	Library	\$MODEL_TECH../altera/verilog/altera

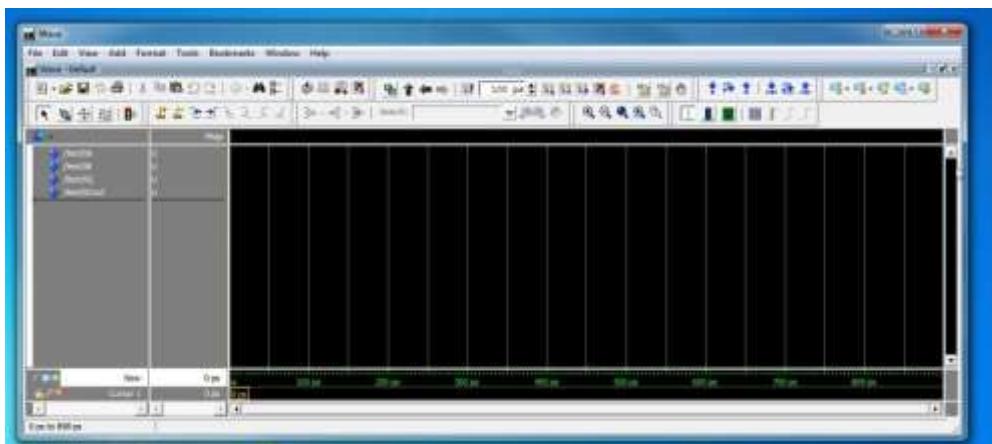
Vy byste tam měli vidět pouze entitu „adder“, já mám přidanou ještě entitu „ch02“, ale té si nevšímejte. Teď musím přeložit samotné testovací zapojení. V menu Compile vyberu Compile, najdu soubor testbench.vhd (bude o dvě úrovně adresáře výš – dialog se otevře v podadresáři „simulation/modelsim“, což je místo, kam si příště své testovací soubory ukládějte) a dvojitým poklepáním soubor přeložím. Pokud je bez chyb, přidá se do seznamu v knihovně „work“:

Name	Type	Path
work	Library	rtl_work
+ E adder	Entity	D:/VHDL/webdemo/ch02/ch02.vhd
+ E ch02	Entity	D:/VHDL/webdemo/ch02/ch02.vhd
+ E test	Entity	D:/VHDL/webdemo/ch02/testbench.vhd
+ rtl_work	Library	D:/VHDL/webdemo/ch02/simulation/m...

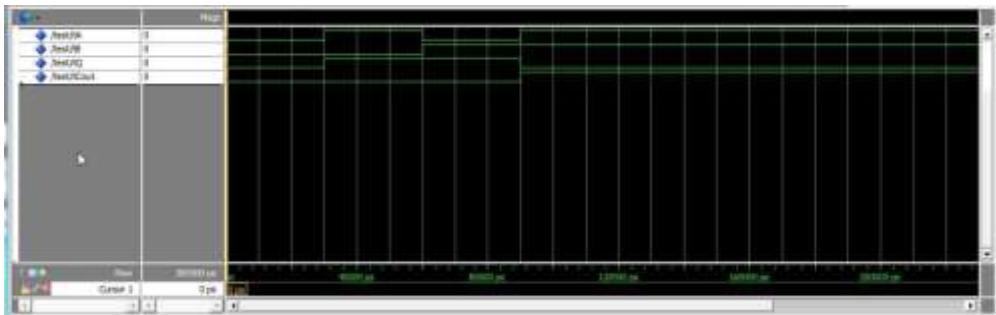
Klikněte pravým tlačítkem myši na „test“ a vyberte „simulate“. Okna se budou chvilku přeskupovat, a nakonec uvidíte něco, co bude velmi podobné následujícímu obrázku:



Vlevo si můžete projít hierarchii celého zapojení (není moc složitá, je to „test“, který obsahuje „UUT“), vpravo pak vidíte signály pro vybranou entitu. U entity „test“ to jsou signály tA, tB, tQ a tCout. Vyberu je myší, kliknu pravým tlačítkem a zadám „Add Wave“ (nebo Ctrl-W). Otevře se okno s průběhy signálů (Wave – takový virtuální logický analyzátor). Kliknutím na ikonku v pravém horním rohu panelu si přepnou Wave ze zobrazení v panelu na samostatné okno. V něm proběhne vlastní simulace.



Nahoře uprostřed vidíte hodnotu „100 ps“ – tedy sto pikosekund. To je krok, po kterém chceme simulovat. Doporučuju přepsat na „100 ns“, protože signály měníme až po 30 ns, a tak bychom se taky nemuseli dočkat. Tlačítkem vpravo od výběru intervalu spustíme jeden simulační běh (nebo též stiskem F9). Na displeji se vykreslí čáry, které udávají průběh signálu – pravděpodobně budou všechny rovné, proto si nejprve pomocí lupy (se znaménkem MINUS) změňte měřítko osy tak, aby bylo vidět alespoň těch 100 ns najednou.



Vidíte, že vše funguje tak, jak má. Signály A a B se mění tak, jak jsme zapsali, a sčítáčka správně nastavuje výstupy Q i Cout.

Testování je při vývoji nezbytná část, a proto jsem ji zařadil hned takhle na začátek. V dalším pokračování se budeme zase věnovat více teorii, ale je dobré vědět, že máte k dispozici nástroj, kterým si můžete otestovat to, co jste se naučili.

(Já vím, to slibované blikání LEDkou to stále ještě není, ale zase uznejte – už to SKORO je, a který jiný jazyk vám umožní si svoje „hello world“ nasmulovat v duchu hesla „takhle nějak by to vypadalo, kdyby se to spustilo“?)

## Komponenty a signály

Už jsme na obojí narazili. Pojdme si nyní tyto pojmy probrat podrobněji. Doufám, že jste část věnovanou testování nepřeskočili. To by byla velká chyba. Ukázali jsme si v ní totiž další dva základní koncepty.

### Komponenty

První z nich je koncept **komponenty**. Elektronické obvody se skládají z celků, které se skládají z menších celků... atd. Například nějaká deska obahuje několik multiplexorů. Multiplexery jsou složené z hradel, hradla jsou složena z tranzistorů.... I ve VHDL je postup, jak nadefinovaný jednodušší obvod použít ve složitějším. Už jsme si ukázali, jak se obvod popisuje, že se skládá z deklarace **entity** a z popisu **architektury**. To jsou pro nás stavební bloky, které můžeme použít v jiných stavebních blocích. Podobně jako v jazyce C v jednom souboru funkci deklarujeme (.h), v jiném

definujeme (.c) a v dalším používáme (.c), tak i ve VHDL musíme v té části, kde entitu použijeme, zopakovat její deklaraci, aby syntetizér věděl, jak entita komunikuje s okolím (o její architektuře nepotřebuje vědět nic). Použije se k tomu postup, při němž zopakujeme deklaraci entity, jen místo „entity“ napíšeme „component“. Tím se z „entity“ stává „komponenta“ pro daný obvod. Entity se zapisují do architektury, mezi hlavičku („architecture X of Y is...“) a začátek definice („begin“). Komponentu pak můžeme použít, vytvořit její „instanci“. Při tomto „instancování“ musíme říct, kam se připojí jednotlivé vstupy a výstupy dané komponenty. Slouží k tomu klíčová slova „port map“ – tedy „mapování portu“. Port je deklarován v entitě, jeho deklarace je zopakována v komponentě, a za slovy „port map“ je v závorce uvedeno, kam se připojuje který výstup. Nebojte, za chvíli bude vše jasnější.

### *Signál*

---

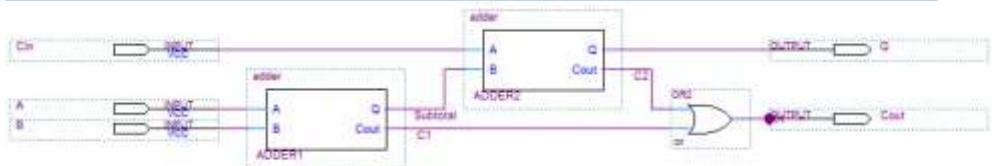
Pouze u těch nejjednodušších obvodů lze připojit vstupy a výstupy komponenty přímo na vstupy a výstupy obvodu. Šlo to například u naší neúplné sčítačky. Tam jsou dvě hradla, XOR a AND, a u obou jsou vstupy připojené k vstupům sčítačky, výstupy k výstupům. Co ale v situaci, kdy chceme připojit (například) vstup jednoho hradla na výstup druhého? Narázíme na to, že pro podobné propojení „nemáme jméno do port mapy“. VHDL proto zavádí koncept **signálu**. Signál je „interní vodič“ – můžete si ho představit jako fyzický vodič, kterým jsou propojeny jednotlivé komponenty v obvodu. Signál má stejné typy jako vstup a výstup z entity, jen nemá určený směr (protože nevede mimo obvod). Dalo by se také říct, že i vstupy a výstupy v entitě jsou specifické signály, které mají určený směr. Můžeme tak hovořit o „vstupní – výstupních signálech“ (až dotedž jsem se tomuto označení bránil, právě proto, aby se nepletly „vstupní signály“ se „signálem“). Signály rovněž umožňují zavést zpětnou vazbu, totiž data z „výstupu“ přivést opět na vstup nějaké komponenty. (Tady ale upozorňuju na jednu záladnost, ke které se vrátím, a ta se jmenuje *latch*...)

Ve VHDL totiž nelze v port map namapovat „výstup z entity“ na „vstup do vnitřní komponenty“.

Dosti teorie. Pojdme k praxi. Popišme si „plnou sčítačku“. Plná sčítačka se od naší neúplné liší tím, že pracuje i se vstupním přenosem (Cin). Má tedy

tři vstupy ( $A$ ,  $B$ ,  $Cin$ ) a dva výstupy ( $Q$ ,  $Cout$ ). Můžeme si zase udělat pravdivostní tabulku a poskládat si sčítáčku z hradel, nebo můžeme zvolit ten přístup, kdy pomocí naší neúplné sčítáčky sečteme  $A$  a  $B$ , a k takto vzniklému mezivýsledku přičteme vstupní přenos. Výsledný přenos je dán přenosem z jednoho nebo druhého sčítání (pokud se vyskytne, bude i na výstupu).

<b>A</b>	<b>B</b>	<b>Cin</b>	<b>Q</b>	<b>Cout</b>
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



Ze schématu je patrné, jak jsou na sebe bloky napojené. Máme pět vstupně – výstupních signálů (Cin, A, B, Q, Cout). Uvnitř jsou tři další spoje: mezi výsledkem první sčítáčky a vstupem B druhé (Subtotal), a pak mezi jednotlivými přenosy a hradlem OR (C1 a C2). Pojďme si tedy napsat plnou sčítáčku. Začneme opět deklarací:

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

entity fulladder is
  port (
    A, B, Cin: in std_logic;
    Q, Cout: out std_logic
  );
end entity;
```

Není tam nic, co by nám bylo neznámé. Entita se jmenuje fulladder a její porty jsou velmi podobné naší sčítáčce, jen je přidaný port Cin. Druhá část bude architektura. Začneme hlavičkou:

```
architecture main of fulladder is
  begin
```

Po hlavičce musí přijít popis použité komponenty. Na rovinu přiznávám, že to je kopie deklarace z entity, přejmenovaná na component.

```
component adder is
  port (
    A, B: in std_logic;
    Q, Cout: out std_logic
  );
end component;
```

Nyní je na místě nadeklarovat si signály. Syntax je:

```
signal jméno[, jméno ...] : typ [:= {výchozí hodnota}];
```

Tedy takto:

```
signal Subtotal, C1, C2: std_logic;
```

Nic víc nepotřebujeme, pojďme popsat chování. Tu část uvozuje, jak už víme, toto:

```
begin
```

Nyní si vytvoříme dvě instance sčítáčky (komponenta adder) a nastavíme propojení:

```
ADDER1: adder port map (A, B, Subtotal, C1);  
ADDER2: adder port map (Cin, Subtotal, Q, C2);
```

Drobná pauza... Všimněte si zápisu. Je to jméno instance, dvojtečka, jméno komponenty, klíčová slova **port map** a v závorce seznam signálů. Ten seznam má kolik položek, kolik položek má port v komponentě adder (tedy čtyři) a ve stejném pořadí, v jakém jsou v komponentě, jim přiřazuju nějaké signály uvnitř nového obvodu. Tedy: ADDER1 je první (levá) sčítáčka. Vstup A je zapojen na vstupní signál A, totéž se vstupem B, výstup Q je připojen na signál („vodič“) Subtotal, tedy mezisoučet, a výstup Cout je připojen na signál C1. Pro druhou instanci sčítáčky platí, že vstup A bere signál Cin, vstup B bere to, co je na signálu Subtotal, výstup Q je připojen na stejnojmenný výstup celého obvodu, no a výstup Cout tvoří druhý signál přenosu C2. Ukážeme si ještě alternativní zápis, kde nezáleží na pořadí. V něm v port map použijeme tvar „port komponenty => místní signál“

```
ADDER1: adder port map (A=>A, B=>B,  
Q=>Subtotal, Cout=>C1);  
ADDER2: adder port map (Q=>Q, A=>Cin,  
Cout=>C2, B=>Subtotal);
```

Takto tedy vypadá strukturální popis architektury. Zbývá už jen logický součet (OR), který vezme C1 a C2 a výsledek pošle na výstup Cout. Můžeme si vytvořit komponentu s funkcí OR, udělat její instanci a pomocí port map určit, jak bude připojena. Ale mnohem snazší je přimíchat trochu data flow:

```
Cout <= C1 or C2;
```

A to stačí. Máme popsané vše, co je v obvodu použité, takže nezbývá než udělat pápá:

```
end architecture;
```

Za domácí úkol si napište testbench pro tuto sčítačku a pomocí ModelSimu si otestujte průběhy signálů.

### Přiřazení signálů

Zatím jsme si ukázali to nejjednodušší přiřazování, kdy nějakému signálu je přiřazena hodnota operátorem `<=`. Na pravé straně může být výraz a syntetizér se ho pokusí převést do ekvivalentní realizace v obvodové podobě. Výraz může obsahovat základní matematické a logické operátory (+, -, AND, OR, NOT...), závorky a další věci, na které jsme zvyklí z programovacích jazyků. **Druhá forma** je podmíněné přiřazení. Jeho tvar je takovýto:

```
Cíl <= {výraz} when {podmínka} [ else
    {výraz} when {podmínka}...] else
    {výraz};
```

Podmínka není nic jiného než výraz, který je vyhodnocen na log. 1, nebo log. 0. Tedy například:

```
Q <= '0' when (A = B) else
    '1';
```

Čteme jako: Q bude 0, pokud A=B, jinak 1. V podstatě je to část naší neúplné sčítačky. Všimněte si, že se porovnávání zapisuje jednoduchým rovníkem, nikoli zdvojeným. Další zdroj častých chyb u programátorů, co přecházejí z „C-like“ světa. Složitější příklad:

```
Q <= '0' when (A = B AND Cin='0') else
    '0' when (A = '0' AND B = '0' AND Cin='1') else
    '1';
```

Opět slovy: Q je nula, pokud A=B a Cin je nulové. Pokud není, tak Q je nula, pokud A i B jsou 0 a Cin je 1. Jinak je Q = 1. U jednobitového výrazu to tak nevynikne, ale u vícebitových, ke kterým se dostaneme příště, bude

ta výhoda patrná. Třetí možná forma je syntaktický cukr pro druhou formu v případě, že se rozhodujeme podle jednoho signálu.

```
with {rozhodovací výraz} select
    Cíl <= {výraz} when {hodnota} [, 
        {výraz} when {hodnota}...][,
        {výraz} when others];
```

Trošku to připomíná známou konstrukci switch-case z programovacích jazyků. Poslední řádek definuje, co se stane, pokud bude výsledek rozhodovacího výrazu jiný než některá z možností (obdoba „default“). U naší plné sčítáčky se například mohu rozhodnout podle signálu Cin, a pokud bude 0, tak výsledek nastavím podle výrazu A=B, pokud je Cin 1, výsledek bude A/=B. (*Bod má ten, kdo si tipnul, že /= znamená „nerovná se“.*) Tedy takto:

```
with Cin select
    Q <= (A XOR B) when '0',
        NOT (A XOR B) when '1';
```

Nemůžu zapsat  $(A = B)$ , protože výsledek porovnání není logická hodnota a nelze jej syntakticky jednoduše na logickou hodnotu převést, proto zapisuju pomocí XOR a NOT XOR.

### *Help I Accidentally Build A Latch*

Jestli vám až do této chvíle připadal latch jako úplně normální součástka, jakých jsou plné katalogy (SN7475 například), tak po téhle podkapitole se váš pohled na něj změní.

Latch je součástka, která má dva vstupy, datový a řídicí. Když je na řídicím log. 1, je obvod průchozí a „co na vstupu, to na výstupu“. Jakmile se změní řídicí vstup na 0, tak obvod drží na výstupu poslední hodnotu před touto změnou. Pamatuje si. Což je docela užitečná funkce, pokud ji potřebujeme použít. V takovém případě o ní víme, VHDL syntetizér správně takovou funkci převede do obvodové podoby, u FPGA od Altery to zabere jeden logický element, a je to v pořádku.

Problém je, když latch vznikne takříkajíc „bez našeho přičinění“. Jak? No, stačí drobnost: Zapomeneme ošetřit všechny možné kombinace na vstupech! Představme si, pro tu úplnou jednoduchost nejjednodušší, že zapisuju neúplnou sčítáčku a zvolím k tomu podmíněné přiřazení „when“.

Správný zápis bude:

```
Q <= '1' when A=B else  
      '0';
```

Jenže co když zapomenu na tu „default“ část, tedy *else*?

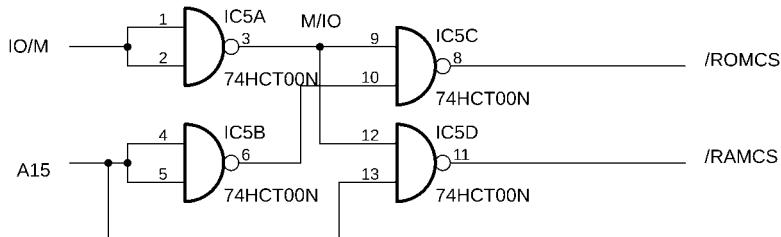
```
Q <= '1' when A=B;
```

Tady to je málo pravděpodobné, ale u složitějších obvodů se to může stát. Myslíte si, že jste ošetřili všechno, nestalo se. Nebo u behaviorálního popisu zapomenete v nějaké větví nastavit nějaký výstup. Co se stane? VHDL si s tím poradí jednoduchou úvahou: *Pokud neřeknete, jakou má mít výstup hodnotu, necháme tam takovou, jaká byla!* To je klasický přístup z programovacích jazyků: Když neřeknu, že se to má změnit, tak se to nemění.

Jenže elektronický obvod nemá nic jako „neměň výstup“, a pokud chci, aby se výstup nezměnil, tak si musím jeho hodnotu někde zapamatovat. Kde? No, zkuste hádat! A aniž byste to chtěli, tak vznikne latch, většinou zcela nadbytečný, a zabírá místo v návrhu. „Omylem vytvořený latch“ je něco jako memory leak, neuvolněný zdroj nebo *omylem postavená police*.

Zkrátka chyba v návrhu, kterou syntetizér sice nějak vyřeší, ale za cenu zbytečného mrhání drahocennými logickými elementy. Většinou to znamená, že jste zapomněli na nějaké přiřazení výstupu za nějakých podmínek. U prostého přiřazení se to nestává, ale jakmile použijete podmíněné přiřazení nebo podmínky jako takové, může se to stát. Nejen že to vygeneruje latch navíc, ale taky to často znamená, že výsledné zapojení nebude fungovat tak, jak má. Proto pozor na takové situace a **vždy výstupům přiřaďte nějakou hodnotu!**

Pojďme se podívat na praktické zapojení. V popisu počítače OMEN Alpha, viz kniha Porty, bajty, osmibity, jsem ukazoval zapojení hradla 7400, které se staralo o správné generování signálů /RAMCS a /ROMCS ze signálů IO/M a A15. Připomeňme si:



Toto zapojení můžeme přepsat zcela jednoduše:

```
library ieee;
use ieee.std_logic_1164.all;
entity alpha is
    port (
        IOM, A15: in std_logic;
        nRAMCS, nROMCS: out std_logic
    );
end;

architecture main of alpha is
signal MIO, nA15: std_logic;

begin
    nA15 <= not A15;
    MIO <= not IOM;
    nROMCS <= MIO nand nA15;
    nRAMCS <= MIO nand A15;
end architecture;
```

Namísto popisu pomocí logických funkcí, tedy „jak to je zapojené“, můžeme zkusit vysvětlit, jak se má obvod chovat. Deklarace entity zůstane stejná, architektura se změní:

```
architecture cond of alpha is
begin
```

```
nROMCS <= '0' when (IOM='0' and A15='0') else '1';
nRAMCS <= '0' when (IOM='0' and A15='1') else '1';

end architecture;
```

Vidíme, že /ROMCS je 0, pokud je A15 = 0 a IOM = 0, jinak je 1, obdobně pro /RAMCS.

Dejme tomu, a to dávám do velkých pomyslných uvozovek, že bychom se rozhodli v počítači Alpha použít místo obvodu 7400 nějaký programovatelný obvod. Je to samosebou nesmysl, ale právě proto píšu: Dejme tomu. Jak bychom postupovali?

Strukturu máme hotovou, funkci nadeklarovanou, syntéza probíhá bez chyb. V tuto chvíli by bylo jen potřeba alokovat piny obvodu pro konkrétní vstupy.

Všimněte si v podokně Tasks (v levém panelu), že se úkol „Compile design“ dělí do několika podúkolů. První je „Analysis and Synthesis“. Součástí tohoto podúkolu je část „I/O Assignment Analysis“. Když si tuto možnost rozbalíte, najdete dvě položky: View Report a Pin Planner. Pin Planner je to, co nás zajímá. Když na tuto položku poklepete, otevře se editor, v němž vidíte pouzdro vybraného obvodu a přiřazení pinů jednotlivým signálům. Zde můžete přiřazení do jisté míry změnit. Píšu „do jisté míry“, protože některé piny mají napevno dané funkce a nelze je přiřazovat. A protože FPGA mívají možnost pracovat hned s několika referenčními napětími, mají různé skupiny vývodů různé možnosti napěťových úrovní atd. Ale obecně platí, že si můžete poskládat vývody tak, jak potřebujete, pokud respektujete daná omezení.

Já mám kit, který obsahuje mimo jiných součástek i několik tlačítek a LED. V dokumentaci jsem našel, že dvě tlačítka jsou na pinech 90 a 91, dvě LED na pinech 1 a 2. Vybral jsem tedy tyto piny a přiřadil jsem je signálům IO/M, A15, /RAMCS a /ROMCS.

Nechal jsem celé zapojení syntetizovat a přes volbu „Program Device“ nahrál konfiguraci do kitu. Pomocí tlačítek jsem si ověřil, že vše funguje, jak má...

## Cvičení

Dekodér pro sedmisegmentovky

## Bit sem, bit tam...

I počítače jsou alespoň osmibitové. Buďme i my vícebitoví!

Až dosud jsme si ukazovali všechno jednabitové: Jednabitová sčítačka s jednabitovými daty, jednabitové signály... Copak VHDL neumí udělat pořádnou sběrnici, třeba datovou, osmibitovou? No, umí. A dokonce hned několika způsoby.

### Vektor

Signál, který je vícebitový, tj. obsahuje několik signálů typu std\_logic, lze ve VHDL zapsat jako vektor. Příklad – osmibitová datová sběrnice bude:

```
signal DBUS: std_logic_vector (7 downto 0);
```

„std\_logic“ se změnilo na „std\_logic\_vector“, a za tímto typem je zapsaný rozsah 7 až 0 (*downto* počítá směrem dolů, *to* směrem nahoru). Tedy DBUS je signál, skládající se z osmi vodičů s typem std\_logic, očíslovaných 7, 6, 5, ... 0. Proč takhle, proč ne 0 .. 7? Zápis je od nejvýznamnějšího bitu k tomu nejméně významnému (od MSB k LSB) a v tomto případě to je tak, že nejvýznamnější je D7. Hodí se to, když někde chcete pracovat s hodnotou tohoto signálu ne v podobě bitového zápisu, ale v podobě čísla.

Jak přiřadíme hodnotu?

```
DBUS <= "00001010"; -- pomocí výčtu bitů  
  
DBUS <= X"0A";  
-- pomocí hexadecimální hodnoty  
  
DBUS <= (1 => '1', 3 => '1', others=>'0');  
-- výčtem hodnot pro konkrétní byty  
-- a hodnoty pro ostatní (nevyjmenované)  
  
DBUS <= (others=>'0');  
-- všechny byty nastavit na hodnotu 0
```

```

DBUS <= (1 to 3=>'1', others=>'0');
-- všechny bity nastavit na hodnotu 0,
-- bity 1 až 3 na hodnotu 1 ("00001110")

DBUS <= ('1', '0', '1', others=>'0');
-- všechny bity nastavit na hodnotu 0,
-- bity 7 a 5 na hodnotu 1 ("10100000")

DBUS <= "0000101Z";

```

Všimněte si důležité věci: Vícebitové hodnoty (vektory) se zapisují v uvozovkách (na rozdíl od jednobitových hodnot v apostrofech). Podobně je tomu i v jazyce C, kde se znak dává do apostrofů, řetězec do uvozovek.

První řádek představuje prosté přiřazení všech bitů, druhý taky, ale se zjednodušeným zápisem v hexadecimální podobě. Třetí řádek používá výčet – v závorce, oddělené čárkami, jsou zapsány dvojice „bit=>hodnota“. Speciální klíč „others“ znamená „všechny ostatní bity, zde nevyjmenované“. Na čtvrtém řádku je ukázáno, jak se tato vlastnost využívá často pro nastavení všech bitů na určitou hodnotu. Pátý řádek modifikuje výčet, syntax „1 to 3“ označuje bity 1 až 3. Na šestém řádku jsou bity zapsány tak jak jdou po sobě. Sedmý řádek pak slouží jako připomenutí toho, že hodnota nemusí být jen 0 nebo 1, ale třeba i „vysoká impedance“, tedy Z.

Jednotlivé bity se odkazují pomocí zápisu s indexem v kulaté závorce (pozor na zvyk z C a spol., kde se píší do hranatých), takže například *DBUS(0)*.

## Čtyřbitová sčítáčka

---

Pokračujme v našem příkladu a sestavme si ze čtyř jednobitových sčítáček jednu čtyřbitovou. Její zapojení je očividné: Dvě vstupní hodnoty A a B budou tentokrát čtyřbitové vektory, totéž výstup Q. Cin je připojen na vstup Cin sčítáčky nejnižšího řádu, Cout na výstup Cout sčítáčky nejvyššího řádu, a zbytek je propojen tak, že přenos z nejnižšího řádu vede do řádu vyššího... atd. Nějak takhle:

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity adder4 is
  port (A, B: in std_logic_vector (3 downto 0);
        Cin: in std_logic;
        Q: out std_logic_vector (3 downto 0);
        Cout: out std_logic);
end entity;

architecture combo of adder4 is

component fulladder is
  port (
    A, B, Cin: in std_logic;
    Q, Cout: out std_logic
  );
end component;

signal C0, C1, C2, C3: std_logic;
begin
  A0: fulladder port map (A(0),B(0), Cin, Q(0), C0);
  A1: fulladder port map (A(1),B(1), C0, Q(1), C1);
  A2: fulladder port map (A(2),B(2), C1, Q(2), C2);
  A3: fulladder port map (A(3),B(3), C2, Q(3), C3);
  Cout<=C3;
end architecture;

```

Deklarace entity je jasná, o té není potřeba diskutovat. Architektura této sčítáčky obsahuje komponentu fulladder, definovanou v minulé kapitole, a čtyři interní signály C0 až C3, pomocí kterých budeme propojovat výstup Cout jedné sčítáčky se vstupem Cin druhé. V těle jsou pak vytvořeny čtyři instance jednobitové sčítáčky, porty jsou namapovány tak, jak jsme si popsali (A na bity sběrnice A, B na bity sběrnice B, Q na jednotlivé bity z Q, do Cin jsou zapojeny výstupy Cout předchozích stupňů, nejvyšší Cout vede ven a nejnižší Cin je připojen na vstup Cin. Takto sčítáčka funguje, teoreticky, bez problémů.

V reálném světě bych se takto zapojené sčítáče raději vyhnul, protože má jednu výraznou nečistotu, a tou je postupný přenos od jednoho stupně k druhému. Zpoždění na jednotlivých hradlech, které se postupně nasčítává, s sebou v důsledku přinese to, že za určitých podmínek při změně vstupních hodnot bude na výstupu po nějaký čas nesprávná hodnota (než změna přenosu „probublá“). Ještě se tomuto problému a jeho řešení budeme věnovat podrobnejí.

Pojďme si ji ale trochu zesložitit.

```
architecture combo2 of adder4 is

component fulladder is
  port (
    A, B, Cin: in std_logic;
    Q, Cout: out std_logic
  );
end component;

signal C: std_logic_vector (4 downto 0);
begin
  A0: fulladder port map (A(0), B(0), Cin, Q(0), C(1));
  A1: fulladder port map (A(1), B(1), Cin, Q(1), C(2));
  A2: fulladder port map (A(2), B(2), Cin, Q(2), C(3));
  A3: fulladder port map (A(3), B(3), Cin, Q(3), C(4));
  C(0) <= Cin;
  Cout <= C(4);
end architecture;
```

Funkčně je zcela ekvivalentní, jen nejsou definované čtyři signály, ale pětibitový vektor C. Programátor možná v tuhle chvíli zajásá, protože objeví v zápisu jednotlivých instancí určitou logickou strukturu. A pokud se těšíte, že budete moci sčítáčky nějak vygenerovat pomocí cyklu, tak jste na správné stopě:

```
architecture gener of adder4 is

component fulladder is
  port (
    A, B, Cin: in std_logic;
    Q, Cout: out std_logic
  );
end component;
signal C: std_logic_vector(4 downto 0);
```

```

begin
adders: for N in 0 to 3 generate
    myadder: fulladder port map (
        A(N), B(N), C(N), Q(N), C(N+1)
    );
end generate;
C(0) <= Cin;
Cout <= C(4);
end architecture;

```

Definujeme si víc sčítaček pomocí konstrukce **for {proměnná} in {rozsah} generate ... end generate;** a uvnitř pracujeme s N jako s normální proměnnou, pomocí které indexujeme jednotlivé byty ve vektoru. Pro rozsah *0 to 3* vzniknou čtyři sčítačky, jejichž porty budou nastaveny naprostě stejně jako v předchozím příkladu.

### *Generické entity*

Pokud jste programátor, přijde vám to přirozené: Proč definovat sčítačku čtyřbitovou, osmibitovou, a pro každou použitou šířku vlastní, když by stačilo definovat obecnou sčítačku N-bitovou, a pak by se při vytváření instancí řeklo, že tahle bude čtyřbitová a tahle šestnáctibitová. Šlo by to?

Šlo, děkujeme za optání. Vezmeme předchozí definici, tu s generátorem instancí, a řekneme, že šířku si uložíme do parametru „wide“. Když bude 4, půjde o čtyřbitovou sčítačku. Všude, kde se vyskytuje trojka, tak ji nahradíme „wide-1“ (třeba ve výrazech „3 downto 0“), kde se vyskytuje čtyřka, tam ji nahradíme „wide“.

Samozřejmě bude potřeba někde ten parametr „wide“ nadeklarovat. Pro deklarace je určena **entita**, a přesně tam přijde deklarace parametru, a to do části **generic()**. Takto:

```

library ieee;
use ieee.std_logic_1164.all;

entity adder_generic is
    generic (wide: integer);
    port (A, B: in std_logic_vector (wide-1 downto 0);
          Cin: in std_logic;

```

```

Q: out std_logic_vector (wide-1 downto 0);
Cout: out std_logic);
end entity;

architecture gener of adder_generic is

component fulladder is
port (
A, B, Cin: in std_logic;
Q, Cout: out std_logic
);
end component;

signal C: std_logic_vector(wide downto 0);

begin
adders: for N in 0 to wide-1 generate
myadder: fulladder port map (
A(N),B(N), Cin, Q(N), C(N+1)
);
end generate;
C(0) <= Cin;
Cout <= C(wide);
end architecture;

```

Parametry jsou zase zapsané v části *generic()* podobně jako vstupně-výstupní signály v části *port*, jako *{jméno}:{typ}[:={default hodnota}]*. Zde je použitý typ *integer*, tedy celé číslo, bez defaultní hodnoty, tj. šířku musíme vždy zadat.

Jak se taková komponenta používá? Velmi podobně jako negenerická. V architektuře musíte uvést deklaraci komponenty, která je shodná s deklrací entity (včetně té části *generic*), a u instance zapíšeme kromě *port map* ještě *generic map*. Stejným způsobem, jakým uvádíme signály pro port, uvedeme i generické parametry. Příklad použití v testovacím zapojení:

```

library ieee;
use ieee.std_logic_1164.all;

entity test4 is
end;

architecture bench of test4 is

component adder_generic is
generic (wide: integer);

```

```

port (A, B: in std_logic_vector (wide-1 downto 0);
      Cin: in std_logic;
      Q: out std_logic_vector (wide-1 downto 0);
      Cout: out std_logic);
end component;

signal tA,tB,tQ: std_logic_vector (3 downto 0);
signal tCout, tCin: std_logic;

begin

tCin <= '0',
      '1' after 15 ps,
      '0' after 20 ps,
      '1' after 45 ps,
      '0' after 50 ps,
      '1' after 75 ps,
      '0' after 80 ps,
      '1' after 105 ps,
      '0' after 110 ps;

tA <= X"0",
      X"3" after 30 pS,
      X"5" after 60 pS,
      X"9" after 90 pS;

tB <= X"0",
      X"8" after 60 pS;

UUT: adder_generic generic map (4) port map
      (tA,tB,tCin,tQ,tCout);
end bench;

```

Generic map nastaví parametr „wide“ na hodnotu 4, port map pak přiřadí porty.



#### Alternativní zápis map

Pokud se vám nelibí pravidlo „dodržet pořadí“, můžete využít zápisu s pojmenovanými parametry, třeba:

```
UUT: adder_generic
```

```

generic map (
    wide=>4
)
port map (
    A=>tA,
    B=>tB,
    Q=>tQ,
    Cin=>tCin,
    Cout=>tCout
);

```

Připomínám, že VHDL ignoruje konce řádků a mezery, takže používejte s klidným svědomím zápis takový, jaký se vám líbí.

### *Aritmetika (s velkým vykřičníkem!)*

Když už jednou ty vektory jsou, tak by bylo fajn mít možnost s nimi pracovat jako s čísly, že? Ukážu vám, jak to jde udělat, a zároveň důrazně varuju, abyste to tak nedělali, a důvod vám prozradím o kousek níž.

Představte si, že abstrahujeme od toho, že signál jsou nějaké bity vedle sebe, a místo toho s nimi pracujeme jako s číselnými hodnotami. Takže osmibitový signál je buď „0 .. 255“, nebo „-128 .. +127“, to podle toho, jestli si ho definujeme jako signed, nebo unsigned. Použijeme další dvě knihovny:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

```

Co tím získáte? Tak například možnost pracovat s čísly typu signed a unsigned s danou šírkou v bitech, a k nim máte definované základní matematické operace. Takto bychom čtyřbitovou sčítáčku nadefinovali tak, že nepoužívá vektory, ale „unsigned (3 downto 0)“, a samotné sčítání by bylo „ $Q \leftarrow A + B + Cin$ ;“ – takhle prosté, protože máme „plus“ definované. Synthesizer si s tím už nějak poradí, a pokud má daný obvod například integrované hardwarové sčítáčky, tak použije je. Ve skutečnosti to je ovšem o něco větší peklo, protože často musíme přetypovávat z unsigned / signed (s

omezeným rozsahem) na typ integer (obecné celé číslo), kód máme zaplácany nejrůznějšími conv\_integer(x) a conv\_unsigned(x,4) (to jako že na šířku 4 bity), a když to chcete simulovat, tak zjistíte, že to, co syntetizér nějak přeloží, to vám simulátor vyhodí, že tomu nerozumí. Například ve výrazu  $(A+B)>15$  (pro zjištění přenosu) tvrdí, že neví, jaký operátor „>“ použít, a tak si vytváříte další signály... Je to možná pěkná vymoženost, ale někdy to opravdu bolí. Každopádně když to budete chtít použít, nepoužívejte std\_logic\_arith! Proč?

### *Arith, SLV, nebo Numeric?*

Co by to bylo za jazyk, kdyby neměl nějakou pasáž, která rozděluje jeho příznivce na dva nesmiřitelné tábory (a dva menší tábory heretiků). Ve VHDL jsme si už ukázali svatou otázku „std\_logic vs std\_ulogic“, ale máme ještě jednu, možná mohutnější, totiž „logic\_arith & std\_logic\_vector vs numeric“. Má to celé historické pozadí, jak někdo navrhнул jeden standard a ostatní byli nespokojení, ne snad proto, že by standard nebyl dobrý, ale protože ho nenavrhl orgán, který standardně standardy standardizuje (IEEE), a tak navrhli jiný standard, který umí defacto úplně totéž, ale není kompatibilní, ovšem aby to nebylo tak jednoduché, tak to celé má dobrý důvod a je mezi tím rozdíl, a pokud znáte HTML, tak vám řeknu, že je mezi tím rozdíl jako mezi *<em>* a *<i>*.

Totiž, ten druhý standard, standardní od IEEE, není kompatibilní se std\_logic\_arith, protože zavádí stejně pojmenované typy. Pokusíte-li se použít obojí najednou, bude zle. Pokud chcete použít čísla, použijte knihovnu **numeric\_std**, tedy na začátku uveděte:

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

Proč tedy všichni nepoužívají numeric\_std? Nechci se tu pouštět do výčtu argumentů, ale na jedné straně jsou ti, kteří hlásají: „Numeric! Je to standard IEEE, takže tím to je dané!“ Na druhé straně jsou ti, kteří říkají: „Ale většina knihoven používá vector, protože tak elektronické obvody fungují! Když použiju numeric, musím přetytovávat... Takže logicky vector a arith“

Mně je nejsympatičtější třetí tábor, který tvrdí, že obojí má svoje místo. Tam, kde se na vícebitový signál pohlíží jako na číslo (třeba právě u sčítáčky), tam patří numeric\_std (ne arith). Tam, kde vícebitový signál nemá rozměr čísla, tam použít std\_logic\_vector (někdy se též setkáte s označením SLV). Například osmibitový vstup multiplexoru, nebo sběrnice pro přerušení IRQ0 – IRQ7 jsou typické případy, kdy neříkáme „přišlo přerušení 8“, ale „přišlo přerušení IRQ3“ – tedy SLV.

Pokud je potřeba převést, tak se nevyhnete přetyповání. Ale vyhněte se std\_logic\_arith, std\_logic\_unsigned a std\_logic\_signed. Tyto knihovny jsou zavržené (deprecated), navíc jsou méně flexibilní než numeric\_std.

## Typy, operátory a atributy

Nadešel čas... Ale nebojte, bude to krátké, výživné, a velmi užitečné.

I ve VHDL máme možnost zapisovat aritmetické operace. V předchozí kapitole jsem ukazoval základní číselné typy, vysvětlil, proč používat numeric\_std a naznačil, že s nimi lze dělat nějaká ta *matika*. Pojdme na to!

### Základní typy ve VHDL

Každý jazyk má nějaké základní typy, s nimiž se dál pracuje. Ve VHDL jich je rovnou několik – nebudu se rozepisovat podrobněji, spíš jen tak shrnu:

Typ	Hodnoty	Knihovna
<b>std_ulogic (sul)</b>	,U‘, ,X‘, ,0‘, ,1‘, ,Z‘, ,W‘, ,L‘, ,H‘, ,-, ,	std_logic_1164
<b>std_ulogic_vector (sulv)</b>	Vektor (pole) hodnot typu std_ulogic	std_logic_1164

<b>std_logic (sl)</b>	jako std_ulogic, resolved (tj. rozhodnuté hodnoty)	std_logic_1164
<b>std_logic_vector (slv)</b>	Vektor hodnot typu std_logic	std_logic_1164
<b>unsigned (uv)</b>	SLV N bitů, rozsah 0 .. 2N-1	numeric_std
<b>signed (sv)</b>	SLV N bitů, rozsah -2N-1.. 2N-1-1	numeric_std
<b>boolean</b>	výčet (true, false)	standard
<b>character</b>	znak ASCII	standard
<b>string</b>	pole znaků	standard
<b>integer</b>	32bitové signed číslo (-2 <sup>31</sup> -1 .. 2 <sup>31</sup> -1)	standard
<b>real</b>	-1.0E38 .. 1.0E38	standard
<b>time</b>	1 fs .. 1 hr (femtosekunda až hodina)	standard

Unsigned a signed čísla jsou i v knihovnách std\_logic\_arith, std\_logic\_unsigned nebo std\_logic\_signed. Neměli byste je používat (jsou *deprecated* a nejsou standard IEEE), ale měli byste o nich vědět (byly dlouho „de facto průmyslový standard“). Podobný případ jsou typy *bit* a *bit\_vector*, které byly nahrazeny std\_logic.

K typům jako boolean, integer apod. existují i jejich vektorové podoby. K typu integer existují i subtypy *natural* (přirozená čísla s nulou, tj. nezáporná) a *positive* (celá kladná čísla).

### ***Konverze***

---

Ve VHDL jsou některé typy převedeny na jiné automatickou konverzí, jiné musíme převádět explicitní konverzí. Mezi ty implicitní patří:

- Konverze mezi std\_logic a std\_ulogic je automatická
- Elementy vektorů „signed“, „unsigned“ a „std\_logic\_vector“ jsou automaticky převáděny na skalární hodnoty std\_logic, std\_ulogic

Explicitní konverze mezi signed, unsigned a vektory používá název typu a závorky:

- slv <= **std\_logic\_vector**(uv);
- slv <= **std\_logic\_vector**(sv);
- uv <= **unsigned**(slv);
- sv <= **signed**(slv);

Explicitní konverze mezi signed, unsigned a integer používá funkce **to\_XXX**:

- int <= **to\_integer** (uv);
- uv <= **to\_unsigned** (int, 8);
- sv <= **to\_signed** (int, 8);

Funkce to\_unsigned, to\_signed vyžadují druhý parametr, který udává velikost výsledného vektoru v bitech.

Konverze mezi vektorem a integerem vyžaduje mezikrok přes unsigned nebo signed.

Někdy není jasné, jak vyhodnotit literál. Například ve výrazu sv + "1010" není jasné, jestli k signed vektoru přičítáme hodnotu 10, nebo -6.

### ***Uživatelské typy***

---

VHDL umožňuje definovat vlastní typy dat, podobně jako Pascal. Používá se klíčové slovo type, zápis je „type {jméno typu} is {popis typu}“.

### Celočíselné typy

Pomocí klíčového slova „range“ určíme rozsah celočíselného typu. Rozsah se musí vejít do rozsahu typu integer (32 bitů). Například

```
type temperature is range 0 to 100;  
type my_val is range -8 to 7;
```

Pokud použijete jen typ „integer“, VHDL si pro něj vyhradí 32 bitů, což bude většinou nehorázné plýtvání. Proto tam, kde to má smysl, omezte rozsah pomocí nějaké výše uvedené definice.

### Výčty

Obdoba typu množina v Pascalu, popř. enum z C. V závorkách je uveden výčet možných hodnot:

```
type bit is ('0', '1');  
type boolean is (false, true);  
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H',  
'-' );
```

### Pole

Pole je, jako v jiných jazycích, struktura obsahující elementy stejného typu. Zápis je „type {jméno typu} is array ({specifikace rozsahu}) of {typ elementů}“.

```
type bit_vector is array (natural range <>) of bit;  
type t1 is array (positive range <>) of integer; -- nemůže  
mít nulový index  
type t2 is array (0 to 3) of integer; -- pole se čtyřmi  
položkami  
type t3 is array (natural range <>) of std_logic; -- defacto  
std_logic_vector  
type t4 is array (1 to 4, 1 to 4) of std_logic; --  
dvourozměrné pole (matice)
```

Specifikace rozsahu je buď přímo zapsaný rozsah (např. „0 to 3“), nebo specifikace toho, jakých hodnot může nabývat. „Natural range <>“ znamená, že je očekávaný rozsah v rámci přirozených čísel. Specifikace může obsahovat více položek oddělených čárkou, pak vznikne vícerozměrné pole. Jako specifikace může být použit i výčet...

## *Záznamy*

Ano, Pascal opět vystrkuje růžky. Záznam (record) je kompozitní typ, který umožňuje do jednoho typu spojit víc různých typů.

```
type {jméno typu} is record
    {jméno prvku} : {typ prvku};
    [{jméno prvku}] : {typ prvku};
    ...
end record;
```

Například:

```
type opcode is record
    code : std_logic_vector(1 downto 0);
    dest : std_logic_vector(2 downto 0);
    src : std_logic_vector(2 downto 0);
end record;

variable data : opcode;
data := ("01", "100", "101");
data.code := "00"
data := (code=>"10", others=>"000");
...
```

Na položky se odkazujeme dobře známou tečkovou notací.

## *Konstanty*

Zapráskat si kód „magickými konstantami“ dokáže každá lama. Člověk s nějakou sebeúctou použije pojmenovanou konstantu.

```
constant pokojova_teplota: temperature := 20;
constant c2: t2 := (1, 2, 4, 8);
```

Za slovem constant je jméno konstanty, za dvojtečkou její typ, a za znaky „:=“ její hodnota.

## Operátory

VHDL nabízí standardní sadu matematických a logických operátorů, jaké jsou běžné i v jiných jazycích. Z toho, co jsem psal výš, je jasné, že jejich použití nebude žádná selanka a že budou muset existovat jasná pravidla pro to, co se stane, když se potkají v jednom výrazu hodnoty různých typů. A protože je VHDL silně typovaný, jsou pro něj dva různě definované typy odlišné, i když třeba oba představují osmibitový vektor.

Operátory	
<b>Exponent, absolutní hodnota</b>	$^{**}$ , abs
<b>Logické</b>	and, or, nand, nor, xor, xnor, not
<b>Multiplikativní</b>	$*$ , $/$ , mod, rem
<b>Aditivní, negace</b>	$+$ , $-$
<b>Spojování, posuny, rotace</b>	$\&$ , sll, srl, sla, sra, rol, ror
<b>Relační</b>	$=$ , $/=$ , $<$ , $\leq$ , $>$ , $\geq$

VHDL základní operátory intenzivně přetěžuje, díky čemuž můžeme například přičítat celá čísla k vektorům (a výsledkem je zase vektor). Pokud máme např. signál „count“, definovaný jako unsigned ( $uv$ ), můžeme napsat „count + 1“... Platí ale některá pravidla:

- unsigned + unsigned = unsigned
- unsigned + integer = unsigned
- integer + unsigned = unsigned
- signed + signed = signed
- signed + integer = signed
- integer + signed = signed

Navíc platí, že „velikost cíle“ (=počet bitů) musí odpovídat „velikosti výrazu“. Nelze tedy přiřadit výsledek součtu dvou čtyřbitových vektorů do pětibitového (i když by to dávalo smysl). Pokud přemýslíte, jak velké jsou které výrazy, tak vězte:

Výraz	Velikost v bitech
„11001010“	Počet číslic v literálu (8)
X“5A“	Počet znaků * 4
A	Velikost vektoru A
A and B	Velikost vektorů A a B
A > B	Boolean
A + B	Velikost většího z vektorů A, B
A + 10	Velikost vektoru A
A * B	Velikost A + velikost B

**Aditivní operátory** (+, -) dávají výsledek o stejné šířce, jako má větší z operátorů. Pokud dojde k přetečení, nejvyšší bit se ztratí.

U **relačních operátorů** je výsledkem vždy typ boolean. Ačkoli to svádí k záměně se std\_logic, není to tak a tyto typy nejsou zaměnitelné.

**Bitové operátory** představují jednak posuny a rotace, jednak operátor spojení **&**. Ten použijeme při skládání kratších vektorů do delšího, např. výše zmíněný problém „převést čtyřbitové unsigned číslo na pětibitové“ můžeme vyřešit jako  $,0 \& A$ . Dva čtyřbitové vektory složíme do osmibitového pomocí  $A \& B$ . Podle konvence se skládají hodnoty tak, jak jsou zapsány za sebou, MSB je vlevo.

K těmto operátorům bych zařadil i operátor „výběru rozsahu“. Například – potřebujeme do osmibitového vektoru D zkopírovat horních 8 bitů šestnáctibitového vektoru A:  $D \leftarrow A (15 \text{ downto } 8)$ . Rozsah vybereme jeho uvedením v závorce za signálem.

Že se vracím ještě k té sčítáčce:

```
signal A, B, Q: unsigned (3 downto 0);
signal subtotal: unsigned (4 downto 0);
signal Cout: std_logic;

subtotal <= ('0' & A) + ('0' & B);
Q <= subtotal (3 downto 0);
Cout <= subtotal (4);
```

Anebo s troškou hackování:

```
subtotal <= A + B + "00000";
```

U **multiplikativních operátorů** platí, že pokud opravdu potřebujeme násobit čísla, použijeme násobení, protože ho syntetizér může umístit do hardwarové násobičky (ve FPGA bývají...) Dělení, modulo a operátor zbytku (rem) bývají hůř syntetizovatelné...

**Posuny** jistě znáte, ale připomenu: Logické (sll, srl) zaplňují volná místa nulami, aritmetické (sla, sra) opakují nejnižší (sla) nebo nejvyšší (sra) bit.

## Atributy

Datové typy ve VHDL mají určité *atributy*, které jsou použitelné pro zjištění podrobností o typu. Atributy se zapisují jako '**ATRIBUT**', tedy apostrof a jméno atributu, a to přímo za hodnotu nebo typ, na který se ptáme. Ukážeme si je na příkladu:

```
signal D: std_logic_vector (7 downto 0);

D'LEFT -- levá hodnota rozsahu, tedy 7
D'RIGHT -- pravá hodnota, tedy 0
D'LOW -- nejnižší hodnota rozsahu (0)
D'HIGH -- nejvyšší hodnota rozsahu (7)
```

```
D'ASCENDING -- jsou hodnoty stoupající (true), nebo klesající  
(false)?  
D'LENGTH -- počet bitů (8)
```

Svoje atributy mají i signály. Kromě výše uvedených, které se vztahují k typu, lze použít například:

```
D'EVENT -- true, pokud došlo k "události" - tedy pokud se  
signál změnil  
D'LAST_VALUE -- předchozí hodnota signálu
```

Používá se např. k detekci náběžné hrany hodin v procesech: *clk'event and clk='1'*

Atributy pomohou nejen u zpracování signálů, ale např. i při definici generických entit.

Nejen typy, ale i operátory a atributy můžeme definovat vlastní.

## Proces

Tak, teď už to začíná trochu připomínat programování. Ale moc se neradujte, elektronické obvody se konstruují, nikoli programují!

Úmyslně jsem se tomu vyhýbal. V úvodu jsem psal, že máme tři možnosti, jak popsat obvod. Strukturní jsme si ukázali (to je to vytvoření instancí komponent a namapování vývodů na signály), i data flow (to je to, kde popisujeme, jak vzniká který signál). Chybí ta třetí... Ale ještě, než se na ni podíváme, tak chci upozornit na jednu důležitou věc.

Představme si, že máme nějaký obvod, třeba AND-OR-INVERT – čtyřvstupový obvod, který provádí operaci  $Q = \text{NOT}((AB) + (CD))$ :

```
signal A, B, C, D, AB, CD, Q: std_logic;  
  
AB <= A and B;  
CD <= C and D;  
Q <= not (AB or CD);
```

Máte to? A teď si představte, že ho zapíšu takto:

```

signal A, B, C, D, AB, CD, Q: std_logic;

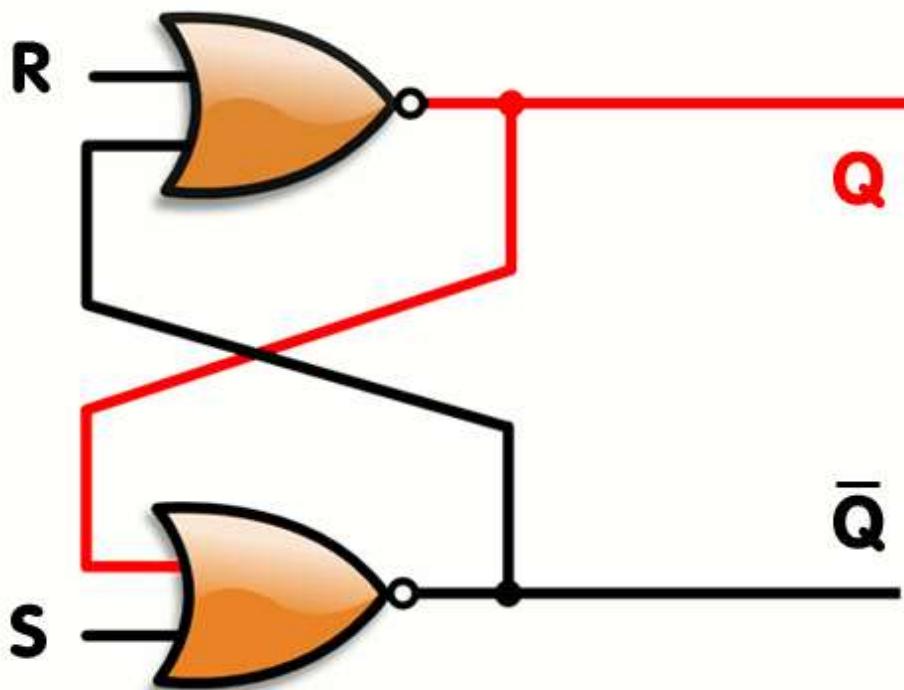
Q <= not (AB or CD);
AB <= A and B;
CD <= C and D;

```

V čem je rozdíl?

Velmi správně: **Není v tom rozdíl!** U programování by v tom rozdíl byl, a pokud jste programátor, máte tendenci ho tam vidět: „Jak můžu přiřadit do Q něco, když u toho ještě neznám hodnotu?“ **Špatně!** Nic nikam nepřiřazuju a žádnou hodnotu neznám „už“ nebo „ještě“. Všechno se děje naráz a výše uvedený zápis není „do AB zadej A and B... a pak do Q zadej...“ Ne. Čtěte to jako „Signál Q vznikne negací součtu signálů AB a CD. Signál AB vznikne součinem...“ atd. *Což mimochodem zjednodušuje vytváření zpětných vazeb.*

Pojďme zpátky k příkladu ze samotného úvodu – totiž ke klopnému obvodu R-S ze dvou hradel NOR.



Ted' už bychom měli vědět, jak ho popsat. Pojďme. Začneme deklarací:

```
entity rsko is
  port (
    R, S: in std_logic;
    Q, nQ: out std_logic
  );
end entity rsko;
```

Strukturní popis bude vypadat nějak takto:

```
architecture struct of rsko is
signal sQ, snQ: std_logic;

component NORGATE
  port (
    A, B: in std_logic;
    Y: out std_logic
  );
end component;

begin
  G1: NORGATE port map (R, snQ, sQ);
  G2: NORGATE port map (S, sQ, snQ);

  Q <= sQ;
  nQ <= snQ;
end architecture;
```

Vidíte, že strukturní popis není „čistý“, museli jsme použít interní signály sQ a snQ, to kvůli pravidlu „výstupní signál nemůže být připojen na vstup“. Komponentu NORGATE zde neřešíme, předpokládáme, že někde jinde máme toto

```
entity NORGATE is
  port (
    A, B: in std_logic;
    Y: out std_logic
  );
end;

architecture main of NORGATE is
begin
```

```
Y <= NOT(A or B);  
end architecture;
```

Dobrá. Jak vypadá data flow popis?

```
architecture dataflow of rsks is  
signal sQ, snQ: std_logic;  
  
begin  
sQ <= NOT (R OR snQ);  
snQ <= NOT (S OR sQ);  
Q <= sQ;  
nQ <= snQ;  
end architecture;
```

### *Behaviorální popis a proces*

Behaviorální popis říká, jak už název napovídá, jak se obvod chová. Tedy nikoli jak vzniká signál Q a nQ, ale „Co se stane, když na vstup R přijde 1? A když přijde na vstup S?“ U R-S obvodu si to dokážeme představit, ale jsou situace, kdy dokážeme říct, co má obvod dělat, ale nechce se nám ho rozkládat na komponenty a popisy signálů. V dalších lekcích takových situací zařijeme ještě spousty.

Klíčovým pojmem behaviorálního popisu je **proces**. A teď dávejte prosím bedlivý pozor: *Proces je výjimečný v tom, že se provádí sekvenčně!* V procesu se postupuje odshora dolů a vykonávají se instrukce tak, jak jdou po sobě. Ale není to tak úplně jednoznačné a vypečou vás hlavně signály.

Z programovacích jazyků můžete mít tendence řešit některé úlohy způsobem „nastavím A na hodnotu 0, pak něco udělám, pak nastavím A na hodnotu 1“. Pokud je A signál, tak *leda houby, velebnosti!* Proces se sice provede sekvenčně, ale pamatujte si, že hodnota se signálům nastaví „až pak, někdy na konci“. Syntetizátoru je úplně jedno, jak šíbujete s hodnotami, jako platnou vezme tu poslední přiřazenou. Na takové operace, které máte pravděpodobně na mysli, se používají *proměnné*.

**Proměnné** jsou velmi podobné signálům, ale:

- definují se lokálně v procesu a platí pouze pro daný proces

- přiřazení není operátorem `<=`, ale operátorem  `$\coloneqq$`
- změna hodnoty je platná okamžitě až do další změny hodnoty

Proces si představme jako „event handler“ z jiných jazyků. Tedy krátkou sekvenci operací, které se provedou, když se něco stane. Každý proces má uveden tzv. *sensitivity list*, tedy seznam signálů, které si musí hlídat, a pokud se některý z nich změní, tak se proces vykoná. Tady je důležité uvědomit si, že ve skutečnosti v obvodu není žádná magická „programovatelná část“. Místo toho syntetizér vymyslí *zapojení*, které se chová tak, jako kdyby probíhal daný proces.

Obecný tvar procesu je:

```
[{jméno procesu}:] process({sensitivity list}) is
  [{deklarace proměnných, podprogramů, typů, konstant,
aliasů, atributů - NE SIGNÁLŮ!}]
begin
  {příkazy}
end process;
```

Jméno je nepovinné, deklarace taky. Pokud používáme nějakou proměnnou (viz výše), deklarujieme ji zde. Klíčové slovo je *variable*.

```
variable a,b: integer;
variable state: integer := 0;
```

Proměnné jsou lokální v daném procesu a jsou nevolatilní, to znamená, že mezi jednotlivými spuštěními procesu uchovávají svou hodnotu. Pokud počítáte s tím, že nějakou hodnotu má mít před prvním spuštěním procesu, použijte deklaraci s přiřazením.

## *Příkazy v procesu*

---

V rámci procesu můžeme provádět (sekvenčně) příkazy. Kromě přiřazení jsou to i jiné konstrukce, známé z programovacích jazyků.

### *Přiřazovací příkaz*

Jen pro pořádek. Můžeme přiřadit hodnotu signálu pomocí  `$\coloneqq$` , nebo proměnné pomocí  `$\coloneqq$` :

## *Podmíněný příkaz*

```
if (podmínka) then
    {příkazy}
end if; -- pozor, nesplést! "endif" neexistuje a vyhodí
spoustu chyb!

if (podmínka) then
    {příkazy}
else
    {příkazy}
end if;

if (podmínka1) then
    {příkazy}
elsif (podmínka 2) then -- nesplést! Není to "elseif", ani
"else if",
                                -- ani "elif", je to "elsif"!
    {příkazy}
... (další else, nebo elseif)
end if;
```

## *Příkaz case*

```
case (výraz) is
    when {hodnoty} =>
        {příkazy}
    [when {hodnoty} =>
        {příkazy} ...]
    [when others =>
        {příkazy}]
end case;
```

Ekvivalentní příkazu „case“ z Pascalu, popř. konstrukci switch-case z C-like jazyků, ovšem s tím rozdílem, že se provedou jen ty příkazy, které jsou u dané hodnoty, není tedy třeba „break“. Může připomínat SELECT, který jsme si popisovali jako jeden z možných tvarů přiřazení. Hlavní rozdíl je v tom, že SELECT je výraz, CASE příkaz. Select se používá u přiřazení, CASE v procesu.

## *Příkaz wait*

Tento příkaz má tři různé formy. V návrhu obvodu použijete jen první dvě, tu třetí využijete v simulačním testbenchi.

První je *wait until* (*podmínka*). Tento příkaz můžeme použít pouze v procesu, který nemá sensitivity list (tj. jako by běžel neustále) a říkáme jím, že se má provádění pozdržet až do chvíle, než bude splněna podmínka.

Druhý tvar je *wait on* (*signál*). Opět můžeme použít pouze v procesu bez sensitivity listu. Pozdrží provádění do změny signálu.

Třetí tvar je *wait for* (*čas*). V testovacím obvodu pro zapojení jím můžeme předepsat čekání na určitou dobu. Vhodné například pro generování hodinových pulsů:

```
process
begin
    wait for 40 ns;
    clk <= not clk;
end process;
```

### *Příkaz loop*

Ano, i ve VHDL existují smyčky.

```
-- věčná smyčka
[{{návěští}:}] loop
    {příkazy}
end loop;

-- smyčka s daným počtem průběhů
[{{návěští}:}] for {identifikátor} in {rozsah} loop
    {příkazy}
end loop;
-- rozsah musí být zapsaný staticky, např. "0 to 7", nelze
zde použít
-- proměnnou nebo signál. "0 to x" nebude přeloženo

-- smyčka s podmínkou na začátku
[{{návěští}:}] while (podmínka) loop
    {příkazy}
end loop;
```

Uvnitř smyčky můžeme použít slovo exit (ekvivalent „break“, tedy vyskočení ze smyčky), bud' v podmíněné větví nebo třeba ve tvaru exit when (*podmínka*). Obdobou příkazu „continue“ je příkaz next– tedy další iterační. Opět možno zapsat jako next when (...)

## Příklad procesu

Vraťme se ještě k našemu klopnému obvodu R-S. Jak ho popsat behavio-rálně? Já zvolil tento způsob:

```
architecture behavioral of rsko is

begin
    process (R,S) is
        variable state: std_logic :='X';
        begin
            if (R='1' and S='0') then
                state:='0';
            elsif (R='0' and S='1') then
                state:='1';
            elsif (R='1' and S='1') then
                state:='X';
            end if;

            Q <= state;
            nQ <= NOT state;

        end process;
end architecture;
```

Tedy: architekturu tvoří jeden proces, který hlídá vstupy R a S a spustí se ve chvíli, kdy se jejich hodnoty změní. V procesu jsem si nadefinoval pro-měnnou *state* – to je pro mne „interní stav klopného obvodu“, tedy pro-měnná, kde mám uloženou zapamatovanou hodnotu. Tato proměnná se propisuje do signálů Q a nQ na konci procesu. Tady je umístění významné! Kdybych dal tyto dva řádky na začátek procesu, nastavila by se hodnota signálů podle původního stavu proměnné!

Ve třech podmírkách vyhodnocuju, co se stalo a jak zareagovat. Buď je nastavený signál R, a pak je interní stav 0, nebo je nastavený signál S, a pak je interní stav 1, nebo jsou nastavené oba vstupy, a pak je interní stav nedefinovaný (X). Pokud jsou oba signály R i S nulové, nijak to neřeším a stav se nemění.

Pomocí tohoto zápisu mohu nadefinovat např. to, že vstup R bude prioritní, tedy když přijdou signály R i S, bude mít „navrch“ signál R a vnitřní stav bude 0...

Na konci článku naleznete testbench, který vyzkouší všechny čtyři architektury a...

*Moment, říkal někdo čtyři?*

Ano, mám připravenou ještě čtvrtou architekturu, která je opět behaviozární, ale v níž jsem místo proměnné použil signál, abych názorně předvedl rozdíl mezi proměnnou a signálem.

```
architecture behavioralS of rsks is
    signal state: std_logic := 'X';

begin
    process (R,S) is
        begin
            if (R='1' and S='0') then
                state <= '0';
            elsif (R='0' and S='1') then
                state <= '1';
            elsif (R='1' and S='1') then
                state <= 'X';
            end if;

            Q <= state;
            nQ <= NOT state;

        end process;
    end architecture;
```

Na první pohled není vidět rozdíl, jen místo proměnné je použitý signál.  
Tipněte si, co se stane?

Testbench je zde:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test is
end;

architecture bench of test is

    signal R, S, Q1, nQ1, Q2, nQ2, Q3, nQ3, Q4, nQ4: STD_LOGIC;

begin
```

```

R <=      '0',
          '1' after 30 NS,
          '0' after 40 NS,

          '1' after 80 NS,
          '0' after 85 NS,

          '1' after 90 NS,
          '0' after 100 NS,
          '0' after 130 NS;

S <=      '0',
          '1' after 15 NS,
          '0' after 20 NS,
          '1' after 60 NS,
          '0' after 70 NS,
          '1' after 90 NS,
          '0' after 100 NS,
          '0' after 130 NS;

UUT1: entity work.rsko(dataflow) port map (R,S,Q1,nQ1);
UUT2: entity work.rsko(behavioral) port map (R,S,Q2,nQ2);
UUT3: entity work.rsko(struct) port map (R,S,Q3,nQ3);
UUT4: entity work.rsko(behavioralS) port map (R,S,Q4,nQ4);

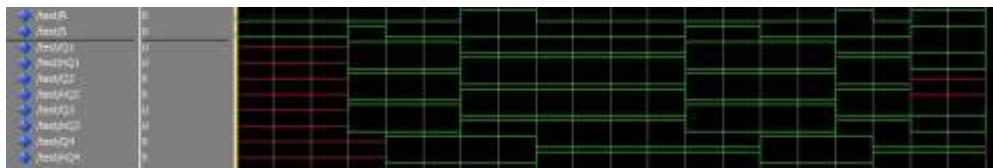
end bench;

```

Na chvílku se u něj zastavím. Všimněte si, že používám čtyři instance entity rsko. Liší se od sebe použitou architekturou. Odkazuju se „plným jménem“, tedy „work.rsko“ (work je jméno knihovny – aktuální projekt je vždy work, rsko je jméno entity) a explicitně říkám, že jde o entitu. Tedy **entity work.xxx(architektura)**. A navíc nikde neuvádím deklaraci komponenty...

**Tip:** Pokud se vám zajídá kopírování port() a generic() z entity do komponenty a říkáte si, že to je hlopost, navíc máte kód zaprášovaný duplicitními zápisu a tak dál, tak vám může tenhle způsob pomoci. Komponentu neinstancujeme jejím názvem, ale zápisem „entity library.name(architecture)“ – pokud se jedná o entitu z téhož projektu, tak použijte speciální jméno knihovny „work“. Uvedení architektury je nepovinné. Můžete taky přesunout entity do knihovny.

Každé instanci připojím stejné vstupy, výstupy připojuju na Qx a nQx.



Vidíme, že signály začínají na 0, pak přijde puls na S, pak na R, pak opět na S, opět na R, a nakonec na obou najednou.

Dataflow architektura (Q1, nQ1) začne správně na stavech X, pak správně reaguje na signály, a na konci, když jsou oba vstupy aktivní, nastaví oba výstupy do log. 0. Je to logické chování, otázka je, nakolik nám takové chování v obvodu vadí.

Strukturní architektura (Q3, nQ3) se chová naprosto stejně.

Behaviorální architektura (Q2, nQ2) se chová správně: korektně zareaguje na nedovolený stav R=S=1 a nastaví správně oba signály na X.

Behaviorální architektura se signálem (Q4, nQ4) se chová velmi podivně... Jako by se změny dělaly správně, ale pozdě! Proč?

Když se nad tím zamyslíte: problém se skrývá v tom, co jsme si tu už řekli jen tak mimořádem o rozdílu mezi proměnnou a signálem: Signál se nastaví „někdy potom“ podle posledního známého přiřazení, zatímco proměnná ihned. Takže když provedu

```
state := '0'; -- state je proměnná!
Q <= state;
```

tak se proměnné state přiřadí hodnota „0“ OKAMŽITĚ, tedy v tu chvíli, kdy je přiřazena, a další příkaz už pracuje s touto hodnotou. Naproti tomu

```
state <= '0'; -- state je signál!
Q <= state;
```

znamená, že se NA KONCI PROCESU přiřadí do Q to, co je během procesu ve state, a do state „0“. Jinými slovy do Q přiřazujeme tu hodnotu state, co měla na začátku procesu. Chování tomu odpovídá: Přijde signál S, zavolá se proces, a na jeho konci se nastaví Q podle úvodní hodnoty state („X“) a state na „1“. Výstup je tedy stále „X“. Za chvíli nato přijde sestupná hrana S, tedy další změna. Opět se vyvolá proces, a na jeho konci se nastaví Q podle úvodní hodnoty state („1“) – state samotné se nemění. A tak dál. Změny se tedy propisují o jedno volání později.

Tohle je další věc, kterou si musíte uvědomovat neustále: **V procesu se jako hodnota signálu bere ta, která byla na začátku.** Po celou dobu běhu procesu je stejná. Všechny změny jako by se zapisovaly do pracovního registru, a do samotného signálu se propíšou až na konci procesu (tedy ta hodnota, kterou přiřadíme jako poslední).

*Tip: Co by se stalo, kdybychom u toho posledního behaviorálního modelu, tj. se signálem, umístili ty dva řádky s přiřazením výstupních signálů mimo proces?*

```
...
end process;

Q <= state;
nQ <= NOT state;
```

### Vylepšená generická sčítáčka

Já vím, už byste chtěli blikat tou LEDkou, a já tu furt se sčítáčkou. Ale tohle je docela dobrý trik...

Vzpomínáte na generickou sčítáčku, které jsme jen zadali počet bitů, a ona se vytvořila přesně podle požadavků? Pojďme ji přepsat tak, že v ní použijeme procesy i aritmetiku.

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

entity adder_generic is
    generic (wide: integer);
    port (A, B: in std_logic_vector (wide-1 downto 0);
          Cin: in std_logic;
          Q: out std_logic_vector (wide-1 downto 0);
          Cout: out std_logic);
end entity;

architecture behavioral of adder_generic is
-- Žádná komponenta ani vnitřní signál
begin
```

```

process(A, B, Cin) is
    variable sum: unsigned (wide downto 0);
    variable sum_vector: std_logic_vector(wide downto 0);
begin

    sum := unsigned('0'&A) + unsigned(B);
    if (Cin='1') then sum := sum+1; end if;
    sum_vector:= std_logic_vector(sum);
    Q <= sum_vector (wide-1 downto 0);
    Cout <= sum_vector (wide);
end process;
end architecture;

```

Používám jeden proces, citlivý na signály A, B a Cin. Uvnitř mám definované dvě pomocné proměnné. V té první sčítám (a zvýším o 1, pokud je přenos), tu druhou použiju pro konverzi na vektor a rozebrání zpět na signály. U sčítání udělám ten trik, co zvýší šířku výsledku o 1 bit. Zbytek je, myslím, naprosto přímočarý a nepotřebuje vysvětlování.

Gratuluji, tímto máme za sebou naprosto nezbytné základy, a od této chvíle už budeme jen probírat praktická zapojení... A možná si i blikneme!

## Hodiny

Tak, nadešel ten okamžik, kdy nám FPGA blikne.

*Hello world!*

Máme kit, na něm LED, kde je problém? Budeme blikat v sekundových intervalech, už víme, jak se dělá proces, takže normálka, ne... LEDku nahodit, počkat sekundu, LEDku vypnout...

Moment, jak jako *počkat sekundu?*

*No, v předchozí kapitole bylo přeci ... „wait for 1s;“*

Nojo, to jsem psal, ale taky jsem psal, že to je pouze pro simulaci!

*Aha, nojo, tak prostě... něco... třeba jako invertor, k němu kondenzátor...*

Nemáme. Teda invertorů máme mnoho, ale nemáme ten kondenzátor.

*A co kdyby se dalo těch invertorů hodně za sebe, tak by to zpoždění vygenerovalo nějaké impulsy a... – a právě cváláte po dráze, postavené z hazardních stavů.*

Nene, nic z toho nepůjde. To, co potřebujeme, je úplně prostý generátor hodin, a musí být někde venku!

Naštěstí na většině kitů je. Na tom mému je taky, kmitá na frekvenci 50 MHz a je připojený na pin 17.

Hodiny máme. Blikání tedy bude jednoduché, stačí podělit ten kmitočet konstantou 50.000.000, neboli padesát milionů, a máme to.

Jak dělit? Tak v zásadě bude potřeba nejdřív někde ty impulsy počítat. Jakmile přijde vzestupná hrana hodin, tak k počítadlu přihodím 1. No a když se to dostane na 50 mega, tak si počítadlo zase vynuluju a změním stav LEDky.

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

entity blink is
    port (
        clk: in std_logic;
        led: out std_logic
    );
end entity;

architecture cntr of blink is

begin
    process (clk) is
        variable counter:integer:=0;
        variable blik:std_logic:='0';
    begin
        if (rising_edge(clk)) then
            counter:= counter + 1;
            if (counter=50000000) then
                counter:=0;
                blik := not blik;
            end if;
        end if;
        led <= blik;
```

```
    end process;  
end architecture;
```

Entita je jasná: Jeden vstupní signál s hodinami, jeden výstupní pro LEDku. Architekturu tvořím behaviorální, v ní se dobře reaguje na změny signálů, a to je přesně to, co potřebuju dělat s těmi hodinami. Takže si vytvořím proces, který bude reagovat na změnu hodin. Použiju dvě proměnné, jednu typu integer, kde si budu udržovat počet cyklů, a pak druhou, kde budu mít stav LEDky. V procesu se nejdřív podívám, jestli přišla vzestupná hrana hodin – k tomu slouží funkce `rising_edge(clk)`. Pokud ano, zvyšuju počítadlo o 1. Pokud dosáhlo hodnoty 50 milionů, tak počítadlo vynuluju a invertuju hodnotu v proměnné „blik“, což je interní stav LEDky. A nakonec, ať se děje, co se děje, pošlu tuhle hodnotu na výstup led.

Proč jsem použil *proměnnou blink?* Nemohl jsem zde použít *signál?* Mohl, ale platilo by to, co jsem psal dřív: změna by se projevila až při další změně hodin. Tady by to asi moc nevadilo, ale radši to nedělám, protože to prostě *není dobré*. Proč jsem nepoužil rovnou `led <= not led`? Protože „led“ je výstupní signál, a ten nemůžu použít ve výrazu. (Ve verzi VHDL2008 to už jde, ale právě tahle feature nepatří mezi ty, které jsou podporované v mojí verzi Quartus II.)

Existuje alternativa k `rising_edge` a zmínil jsem ji v pasáži o atributech: `if (clk'event and clk='1')` Tedy pokud došlo k události na signálu CLK, a zároveň je ted signál CLK = '1'... pak ta událost logicky musela být vzestupná hrana...

Jestli se rozhodnete tentle kód opravdu vyzkoušet, nastavte si – pokud máte stejný kit – signál LED na pin 7 a signál CLK na pin 17.

### *Alternativní blikání*

Napadla mě ještě alternativa, při které si architekturu rozdělím na dvě části, na asymetrickou děličku 1:50000000 (má nestejně délky pulsů), a na děličku 1:2. Z první půjde signál Hz1, na který bude navěšena druhá. Ta bude používat jednobitový signál ff (jako že flip-flop), který při každé náběžné hraně zneguje. Mimo tento proces si napojím výstup LED na signál ff. (Za domácí úkol si zkuste odvodit, proč tady mít signál nevadí, a v předchozím by to vadilo).

```

architecture cntr2 of blink is

signal Hz1: std_logic:='0';
signal ff: std_logic:='0';
begin

process (clk) is
variable counter:integer:=0;
begin
    if (rising_edge(clk)) then
        counter:= counter + 1;
        Hz1<='0';
        if (counter=50000000) then
            counter:=0;
            Hz1<='1';
        end if;
        end if;
    end process;

process(Hz1) is
begin
    if (rising_edge(Hz1)) then
        ff <= not ff;
    end if;
end process;

led <= ff;

end architecture;

```

V architektuře jsou dva procesy, jeden reaguje na změnu clk, druhý na změnu Hz1.

Když jsem to psal poprvé, udělal jsem chybu. V prvním procesu jsem nastavoval defaultní hodnotu Hz1 na '0' mimo podmítku s clk. Nějak takto:

```

begin
Hz1<='0';
if (rising_edge(clk)) then
    counter:= counter + 1;
    if (counter=50000000) then
        counter:=0;
        Hz1<='1';
    end if;

```

```
    end if;  
end process;
```

Syntetizér to odmítl přeložit s tím, že signál Hz1 nijak neudržuje svoji hodnotu mimo to zpracování hodin. Což je trochu kryptické sdělení a netušil jsem, co po mně chtějí. Odpověď je:

„Pokud máte podmínku, která je závislá na události nějakého signálu (typicky náběžné a sestupné hrany časového signálu), dbejte, aby signály měly přiřazenou hodnotu ve všech větvích vnořeného if.“

Každopádně jsem díky tomu vytvořil další alternativní strukturu, která nemění signál Hz1 v podmínce, závislé na clk:

```
process (clk) is  
    variable counter:integer:=0;  
begin  
    Hz1<='0';  
    if (rising_edge(clk)) then  
        counter:= counter + 1;  
    end if;  
  
    if (counter=50000000) then  
        counter:=0;  
        Hz1<='1';  
    end if;  
  
end process;
```

S ní už problém nebyl.

### *Ještě alternativnější blikání*

Ono se číslý, které v desítkové soustavě vypadají dobře (třeba „50000000“) vlogických obvodech špatně dělí. To spíš 2, 4, 8, 65536 nebo 33554432 (což je  $2^{25}$ , kdybyste to chtěli spočítat). Pokud nechceme přesně sekundu, ale „plus minus něco tak aby to bylo okem vidět“, tak použijte prostý binární čítač, kde budete načítat pulsy hodin, no a LEDka bude jeho 24. bit, například.

```

architecture qd of blink is

signal counter: std_logic_vector (25 downto 0):=
(others=>'0');

begin

process (clk) is
begin
  if (rising_edge(clk)) then
    counter <= std_logic_vector(unsigned(counter) + 1);
  end if;
end process;

led <= counter(23);

end;

```

## Klopné obvody, registry a další...

Když už jsem to minule načal a teď jsme si tu zavedli koncept časového signálu, pojďme se podívat na některé základní klopné obvody.

### *Klopný obvod D*

Tento klopný obvod má dva vstupy, D a C, a výstup Q. Když přijde vzestupná hrana signálu C (Clock), zkopíruje se na výstup Q hodnota vstupu D (Data). V ostatních případech si výstup Q udržuje předcházející stav bez ohledu na vstupy.

```

process (C) is
begin
  if (rising_edge(C)) then
    Q<=D;
  end if;
end process;

```

A takhle to je, doslova, nic víc není potřeba.

### *Klopný obvod D s asynchronním nulováním a nastavením*

Tento obvod odpovídá funkcí předchozímu. Navíc má vstupy R a S. Pokud přivedeme 1 na vstup R, zapíše se hodnota '0', pokud na vstup S, zapíše se hodnota '1'. Prioritu má vstup R. Vstupy R a S jsou asynchronní, tzn. změna se projeví hned a nečeká se na hodiny.

```
process (C, R, S) is
begin
    if (R = '1') then
        Q <= '0';
    elsif (S = '1') then
        Q <= '1';
    elsif (rising_edge(C)) then
        Q <= D;
    end if;
end process;
```

Stejný vzor můžete použít všude, kde se míchají asynchronní a synchronní vstupy. Nejprve v podmínkách ošetříte asynchronní, a nakonec si jednu podmínkovou větev vyhradíte pro synchronní operace.

Pokud chcete drsnější, „hackerštější“ podobu, co třeba takto?

```
Q <= '0' when R = '1' else '1' when S = '1' else D when
rising_edge(C);
```

### *Osmibitový registr*

Nebude to o moc složitější než registr D. Ve skutečnosti jsou tyto registry vlastně jen vícenásobné registry D. Ve VHDL to díky vektorům zapíšeme úplně stejně. Pojďme si ale ukázat, jak implementuju funkci „povolovačího vstupu“, tj. hodnota se zapíše jen tehdy, pokud je vstup E (Enable) v log. 1.

```

signal C, E: std_logic;
signal D, Q: std_logic_vector (7 downto 0);

process (C) is
begin
    if (rising_edge(C)) then
        if (E = '1') then
            Q <= D;
        end if;
    end if;
end process;

```

Přibyla jedna podmínka. Všimněte si, že proces není citlivý na signál E – a je to v pořádku. Jediné, co změní stav obvodu, je signál C, a proto je proces závislý pouze na tomto signálu. E je pomocný signál, a jeho změna se na stavu obvodu nijak neprojeví.

### *Osmibitový posuvný registr se synchronním vstupem*

Posuvné registry slouží k serializaci a deserializaci dat. Tento typ se synchronním vstupem můžeme použít k serializaci osmibitového čísla. Což se může hodit například při implementaci sériového rozhraní. Posuvný registr má bitový vstup Din, bitový výstup Dout, vstup hodin C, osmibitovou vstupní bránu D a řídicí vstup Load. Uvnitř je osmibitový registr. Funkce je taková, že při každém hodinovém pulsu se posunou bity o jednu pozici doleva, nejvyšší bit je „vytlačen“ na výstup Dout a do nejnižšího bitu je zkopirována hodnota Din. Pokud je při náběžné hraně signál Load = ,1‘, tak se do registru zkopiruje obsah na datových vstupech D.

```

process (C) is
variable temp: std_logic_vector (7 downto 0):="XXXXXXXX";
variable msb: std_logic := 'X';
begin
    if (rising_edge(C)) then
        if (Load = '1') then
            -- Load je 1, takže nahrajeme nový obsah
            -- Dout se nemění
            temp := D;
        else -- Load je 0, takže se posouvá
            msb := temp(7);

```

```

        temp := temp (6 downto 0) & Din;
    end if;
end if;
Dout <= msb;
end process;

```

Pro zajímavost si zkuste nasimulovat testbench pro tento registr:

```

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

entity test is
end;

architecture bench of test is
signal clk: std_logic:='0';
signal Load, Dout:std_logic;

component shiftreg is
port (C, Din, Load: in std_logic;
      D: in std_logic_vector (7 downto 0);
      Dout: out std_logic
);
end component;
begin
process -- generovani hodin
begin
  wait for 50 ps;
  clk <= not clk;
end process;

Load <= '0',
      '1' after 290 ps,
      '0' after 410 ps;

UUT: shiftreg port map (clk, '0',Load, "10100101", Dout);

end;

```

V testbenchi jsem použil proces na generování hodin, který využívá konstrukci *wait for*.

## Dekodér 1 z 8

Též známý jako 3205 nebo 74138. Tedy ne úplně, vynecháme povolovací vstupy. Namísto procesu použijeme čistě kombinační přístup data flow, v podstatě pravdivostní tabulkou:

```
signal D: std_logic_vector (2 downto 0);
signal Q: std_logic_vector (7 downto 0);

with D select
  Q  <= "11111110" when "000",
    "11111101" when "001",
    "11111011" when "010",
    "11110111" when "011",
    "11101111" when "100",
    "11011111" when "101",
    "10111111" when "110",
    "01111111" when "111",
    "11111111" when others;
```

## Multiplexor 4 na 1

Obsahuje dvoubitový řídicí vstup Sel, čtyři vstupy A, B, C, D a výstup Q. A udělejme si ho třeba generický, s libovolnou šířkou datové sběrnice!

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

entity mux_g is
  generic (bits:natural);
  port (
    sel: in std_logic_vector (1 downto 0);
    A, B, C, D: in std_logic_vector (bits-1 downto 0);
    Q: out std_logic_vector (bits-1 downto 0)
  );
end entity;
architecture main of mux_g is
```

```

begin
with sel select
  Q <= A when "00",
  B when "01",
  C when "10",
  D when "11",
  (others=>'0') when others;

end architecture;

```

## *ALU*

---

neboli Aritmeticko-Logická Jednotka, ve starší české literatuře tedy ALJ. Ne, nebojte se, nebudeme si ukazovat, jak sestavit aritmetickou jednotku, to až někdy příště. Tentokrát jen taková zajímavost...

Představte si, že máte obvod, kde je osm vstupů A, B, C, D, E, F, G a H a dvoubitový řídicí vstup Sel. Podle hodnoty Sel je na výstupu součet dvou vstupních signálů, a to takto:

Sel	Výstup
00	A+B
01	C+D
10	E+F

Sel	Výstup
11	G+H

Podle této specifikace je zapojení takové, že máme čtyři sčítáčky ( $A+B$ ,  $C+D$ ,  $E+F$ ,  $G+H$ ) a multiplexor, který vybírá jeden z výsledků. Ovšem to není rozhodně optimální řešení (4 sčítáčky, 1 multiplexor). Optimální řešení je použít dva multiplexery ((A, C, E, G) a (B, D, F, H)) a jejich výstup poslat do jediné sčítáčky, která je tak sdílená pro všechny čtyři operace.

Obecně je dobrý zvyk co nejvíce sdílet právě aritmetické operace, porovnávání a podobné náročnější věci. Velmi snadno se stane, že v kódu napíšeme velmi podobná sčítání do různých větví podmíněných příkazů (například ve stavovém automatu). Řešení je oddělit procesy, vyhradit si jeden speciální pro náročné „sdílené zdroje“...

## Funkce, procedury, balíčky

Ano, zase se posuneme od drátů k trošku vyšším abstrakcím.

### Funkce

Možná vás napadlo, že by bylo dobré některé opakovány operace v procesech na definovat nějak obecněji, tak, aby byly znovupoužitelné, abyste nemuseli psát kód copy-and-paste, což je vždycky cesta do pekel. Kdyby tak VHDL mělo funkce, co? A vidíte, má je!

```
FUNCTION {jméno funkce} [{seznam parametrů}]
RETURN {typ návratové hodnoty} IS
{... deklarace ...}
BEGIN
{... příkazy ...}
RETURN {výraz};
```

```
END [FUNCTION] {{jméno funkce}};
```

Seznam parametrů je v závorce a je nepovinný. Na konci je nepovinné klíčové slovo FUNCTION a jméno funkce. Ale doporučuju, jako už několikrát, zvyknout si psát END FUNCTION. Syntax zápisu je podobná jazyku C, a ještě podobnější jazyku Pascal, takže pokud znáte Pascal, nepřekvapí vás nic.

Funkci můžete vytvořit v deklaračním bloku u ARCHITECTURE, ENTITY, nebo PROCESS. Popřípadě i v deklaračním bloku jiné funkce.

Funkce je podobná procesu v tom, že má rovněž deklarační a příkazovou část, a že se příkazy v těle vykonávají sekvenčně. Na rozdíl od procesu má funkce návratovou hodnotu (vždy jednu jedinou!) a volitelné parametry.

Seznam parametrů je podobný deklaracím v bloku ARCHITECTURE – parametry jsou definované jako SIGNAL, nebo jako CONSTANT (VARIABLE není povolen), a podle toho se budou ve funkci chovat. Všechny mají typ „IN“, tedy směr dovnitř, do funkce.

### *Assert*

Píšu „volitelné parametry“, takže by bylo dobré je zkонтrolovat. Opět připomínám, že, stejně jako proces, se ani funkce „nespouští“ ve FPGA, místo toho syntetizér „zadrátuje“ algoritmus do obvodů, takže je možné udělat některé statické kontroly přímo ve funkci. Například zkontovalovat, jestli mají parametry správné... ehm... parametry (jako že *vlastnosti*). Slouží k tomu konstrukce ASSERT.

```
ASSERT {podmínka}
  [REPORT {hlášení}]
  [SEVERITY {závažnost}];
```

Pokud je podmínka splněna, nestane se nic. Pokud není splněna (=FALSE), jste na to upozorněni. Můžete pomocí „REPORT“ popsat, co se stalo – do hlášení můžete zahrnout i hodnoty proměnných či signálů a spojit je do jednoho řetězce spojovacím operátorem &. Pomocí SEVERITY můžete sdělit závažnost hlášení. NOTE a WARNING nezastaví proces syntézy, ERROR nebo FAILURE jej zastaví.

Řekněme, že funkce dostane dva vektory (A a B) a něco s nimi provede, to teď není podstatné, podstatné je, že je potřeba, aby oba vektory měly stejný počet bitů. Jako první příkaz tedy uvedeme:

```
ASSERT (a'LENGTH = b'LENGTH)
REPORT "Signály mají rozdílnou délku!"
SEVERITY FAILURE;
```

Porovnáme tedy délku obou vektorů (pomocí [atributu](#) 'LENGTH), a pokud není stejná, vypisujeme hlášení a zastavujeme syntézu – chyba je natolik závažná, že nelze pokračovat dál.

Pomocí assert můžeme vytvořit i testovací výpisy – ASSERT FALSE REPORT „...“ Pokud chcete do výpisu zahrnout hodnotu nějaké proměnné nebo signálu, můžete, ale musíte ji nejprve převést na řetězec. K tomu slouží atribut 'IMAGE – ten se neváže ke konkrétní proměnné, ale k typu, a používá se např. takto: INTEGER'IMAGE(hodnota). Některé typy (std\_logic\_vector) nemají 'IMAGE, je proto potřeba je nejprve přetypovat na integer.

### [Volání funkcí](#)

Tady není nic nezvyklého a zapisuje se tak, jak byste čekali:

```
FUNCTION moje_funkce (a, b: std_logic) RETURN std_logic; ...

Q <= moje_funkce (v1, v2);
Q <= moje_funkce (a=>v1, b=>v2);
Q <= moje_funkce (b=>v2, a=>v1);
```

### [Procedury](#)

Stejně jako v Pascalu se rozlišují funkce (vrací 1 hodnotu) a procedury (nevrací nic), tak i ve VHDL se rozlišují funkce (vrací 1 hodnotu) a procedury (nevrací nic, ale mohou měnit parametry). Syntax je podobná funkcím, odpadá RETURN, a seznam hodnot může obsahovat kromě konstant a signálů i proměnné. Navíc mohou být parametry deklarované jako IN, OUT nebo INOUT.

```
PROCEDURE {jméno} [({seznam parametrů})] IS
{deklarace}
BEGIN
```

```
{příkazy}  
END [PROCEDURE] [{jméno procedury}]
```

Třeba:

```
PROCEDURE test (SIGNAL a,b: IN std_logic; SIGNAL q: out  
std_logic) IS  
BEGIN  
q <= a AND b;  
END PROCEDURE
```

Volání je podobné jako u funkcí, jen se používá samostatně, nikoli ve výrazu (protože nevrací hodnotu).

### *Přetěžování*

VHDL jako silně typovaný jazyk umožňuje velmi snadnou implementaci přetěžování funkcí. Kterou definici použije, to se rozhodne podle typu parametrů. Například balíček numeric\_std přetěžuje funkci „+“ (tedy operátor sčítání) rovnou šestkrát:

```
FUNCTION "+" (L, R: UNSIGNED) RETURN UNSIGNED;  
FUNCTION "+" (L, R: SIGNED) RETURN SIGNED;  
FUNCTION "+" (L: UNSIGNED; R: NATURAL) RETURN UNSIGNED;  
FUNCTION "+" (L: NATURAL; R: UNSIGNED) RETURN UNSIGNED;  
FUNCTION "+" (L: INTEGER; R: SIGNED) RETURN SIGNED;  
FUNCTION "+" (L: SIGNED; R: INTEGER) RETURN SIGNED;
```

Příklad, v němž si přetěžíme operátor „plus“ pro sčítání „slv“ (std\_logic\_vector), najdete na konci kapitoly.

### *Balíčky*

Stejně jako má Pascal své Unity a C svoje header soubory (no, zas tak stejné to není...), tak má i VHDL koncept balíčků. Už je používáme – to je

to „`use ieee.numeric\std.all;`“ Čtěte jako „Použij balíček `numeric\all` z knihovny `ieee`, a z něj vezmi všechno“. Zjednodušuje to a usnadňuje znovu použitelnost některých konstrukcí (funkcí, procedur, vlastních typů atd.)

Balíček (Package) se skládá ze dvou částí, z deklarace obsahu (Package) a z vlastních definic (Package body).

```
PACKAGE {jméno balíčku} IS
{... deklarace ...}
END [PACKAGE] [{jméno balíčku}];

---
[PACKAGE BODY {jméno balíčku} IS
 [{definice funkcí a procedur}]
 [{definice konstant, pokud nebyly definovány v hlavičce}]
END [PACKAGE BODY] [{jméno balíčku}];
```

Package body je nepovinné, a pokud balíček obsahuje např. jen deklarace typů, je zbytečné.

V části Package jsou uvedeny pouze deklarace. Tedy deklarace typů, konstant, signálů, aliasů, funkcí, procedur a dalších věcí, na které ještě přijde řeč. Pokud deklarujeme funkci či proceduru, tak podobně jako v C uvedeme pouze její hlavičku (tj. část až do klíčového slova IS) a ukončíme středníkem:

```
PACKAGE muj_balicek IS
  TYPE matrix IS ARRAY (3 to 0, 3 to 0) of std_logic;
  SIGNAL pole: matrix;
  CONSTANT max: integer := 255;
  FUNCTION getbit (SIGNAL a,b: std_logic_vector(1 downto 0))
RETURN std_logic;
END PACKAGE;
```

V této deklaraci se říká, že balíček obsahuje typ `matrix`, jeden signál jménem „pole“ s typem `matrix`, celočíselnou konstantu `max` s hodnotou 255 a funkci `getbit`, která přijímá dva dvoubitové signály a vrací jednobitovou hodnotu.

Vlastní definice chování funkce `getbit` je uvedena až v části Package body. Zde je uvedena kompletní definice funkce, jak jsme si ukázali výš.

Konstanty mohou být „odložené“, to znamená, že jsou v Package pouze deklarované (CONSTANT max:integer;) a hodnota jim je přiřazena až v Package body (CONSTANT max: integer:=255;)

Balíček použijeme pomocí klíčového slova „use“. Pokud je definovaný v rámci aktuálního projektu, bude mít jeho použití tvar „use work.muj\_balicek.all;“ Aktuální projekt vždy odpovídá knihovně „work“.

A zde je slíbená ukázka: Balíček „muj\_balicek“, který obsahuje funkci „+“, přetíženou pro dva vektory o stejné velikosti. K tomu i dva užitečné atributy, které jsem zatím nezmínil, totiž 'RANGE a 'REVERSE\_RANGE. Oba udávají rozsah vektoru, jeden tak, jak byl vektor definován, druhý v obráceném pořadí.

Příklad: Vektor je a: std\_logic\_vector (7 downto 0). Pak a'RANGE odpovídá (7 downto 0), a'REVERSE\_RANGE je (0 to 7).

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

package muj_balicek is
    function "+" (a,b:std_logic_vector) return
    std_logic_vector;
end package;

package body muj_balicek is
    function "+" (a,b:std_logic_vector) return
    std_logic_vector is
        variable result: std_logic_vector (a'RANGE);
        variable carry: std_logic:='0';
    begin
        for i in result'REVERSE_RANGE loop
            result(i) := a(i) xor b(i) xor carry;
            carry := (a(i) and b(i))
                    or (a(i) and carry)
                    or (b(i) and carry);
        end loop;
        return result;
    end function;
end package body;
```

Upozornění: Funkce nefunguje při sčítání vektoru a literálu, pro takové použití by bylo potřeba ji upravit a obě vstupní hodnoty nejprve převést na

stejný typ. Neřeší ani rozdílné délky vstupních vektorů. První problém vyřeší například alias, což je způsob, jak si „přetypovat“ signál pod jiným jménem:

```
function "+" (a,b:std_logic_vector) return
std_logic_vector is
    variable result: std_logic_vector (a'HIGH downto 0);
    variable carry: std_logic:='0';
    alias aa: std_logic_vector (a'HIGH downto 0) is a;
    alias bb: std_logic_vector (b'HIGH downto 0) is b;
begin
    for i in 0 to result'HIGH loop
        result(i) := aa(i) xor bb(i) xor carry;
        carry := (aa(i) and bb(i))
            or (aa(i) and carry)
            or (bb(i) and carry);
    end loop;
    return result;
end function;
```

Úpravu, ve které správně vezmete velikost výsledku jako „větší z velikostí a, b“ a tomu odpovídajícím způsobem ošetříte vlastní sčítání, nechám na vás.

## Digitálně-analogové převodníky

Už jsme si zablikali, tak co si teď ukázat něco dalšího? Co třeba neblikat tak natvrdo, ale tu LEDku tak jako pomalu rozsvěcit...

Pokud je blikání LEDkou obdobou Hello world, tak je tahle úloha obdobou „PRINT 1+1“. Pokud jste si někdy hráli s Arduinem, tak víte, že LEDka je připojená na digitální výstup, který jaksi nemá nic jiného než „plný jas – tma“, a že se tedy pomalé rozsvěcení řeší pomocí PWM.

### PWM

PWM, neboli Pulsně-šířková modulace (Pulse Width Modulation) je způsob, jak na binárním výstupu (0 / 1) nasimulovat analogový signál (0 .. 1).

V té „správné“ podobě se používají digitálně-analogové převodníky (DAC), buď integrované, nebo v jednoduché podobě [R-2R síť](#), kde vícebitový digitální signál převádíme na analogový. Pokud je situace vhodná, lze použít ale jednoduššího způsobu, a tím je právě PWM.

Princip PWM je jednoduchý: mějme vstupní hodnotu, řekněme osmibitovou, a nazveme si ji D. K ní si uděláme čítač hodinových pulsů, rovněž osmibitový, který bude počítat od nuly k 255 a pak znova od nuly. Dokud je hodnota čítače menší než D, bude na (jednobitovém) výstupu 1, jakmile je hodnota vyšší než D, bude na výstupu 0. Jinými slovy: pokud máme hodiny s frekvencí  $f$ , bude na výstupu obdélníkový signál s frekvencí  $f/256$ , který má ale různou střídu, tedy dobu log. 1 a log. 0. Kdybychom si ten interval rozdělili na 256 částí, tak signál bude po D částí v log. 1 a po (256-D) částí v log. 0. Mění se tzv. *plnění (duty)*. Pokud je frekvence  $f$  dostatečně vysoká (a co je „dostatečně“, to závisí na okolnostech), dá se na signál nahlížet tak, jako by se jeho hodnota měnila spojitě mezi 0 a 1 po 256 krocích. Z definice vyplývá, že pokud D=0, tak je na výstupu stále 0 (0/256), pokud D=255, je na výstupu 255x log. 1 a 1x log. 0 (255/256), tedy 99,6 %. Pokud chceme mít maximální plnění 100 %, pak je potřeba buď zvětšit velikost D o 1 bit, nebo zkrátit čítač o 1 (tedy nejvyšší hodnota bude „11111110“).

Pokud jde o blikání LED, je vhodná frekvence taková, která je větší, než je lidské oko schopno rozpoznat, tj. nějakých 24 Hz. Při osmibitovém PWM pak musí být vstupní frekvence alespoň  $24 * 256 = 6,144$  kHz. V praxi se používají frekvence okolo 100 kHz. Při řízení servomotorů je frekvence PWM signálu 50 Hz, tedy vstupní frekvence pro osmibitový převodník bude  $50 * 256 = 12,8$  kHz. Pokud budeme generovat audio signál, kde je nejvyšší frekvence okolo 22 kHz, a použijeme osmibitový PWM, musíme k vytváření signálu použít frekvenci 5,632MHz.

Asi už není, co víc k tomu dodat, takže zde je kód čtyřbitového PWM:

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

entity PWM is
    port (
        Q: out std_logic;
        D: in unsigned(3 downto 0);
        Clk: in std_logic
    );
end entity;
```

```

architecture main of PWM is
signal cnt: unsigned(3 downto 0) := (others=>'0');
begin
    process (Clk) is
        begin
            if (rising_edge(Clk)) then
                cnt <= cnt+1;
            end if;
        end process;

    Q <= '1' when D>cnt else '0';
end architecture;

```

Je v něm víceméně doslova zapsán algoritmus, popsaný v předchozích odstavcích. Udělal jsem si i druhou architekturu, která implementuje zkrácení čítače o 1:

```

architecture best of PWM is
signal cnt: unsigned(3 downto 0) := (others=>'0');
begin
    process (Clk) is
        begin
            if (rising_edge(Clk)) then
                if  cnt<"1110" then
                    cnt <= cnt+1;
                else cnt <= "0000";
                end if;
            end if;
        end process;

    Q <= '1' when D>cnt else '0';
end architecture;

```

a s malou úpravou je možné obojí nadefinovat jako generickou entitu s proměnnou šířkou:

```

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

entity PWM_G is

```

```

generic (bits: natural);
port (
    Q: out std_logic;
    D: in unsigned(bits-1 downto 0);
    Clk: in std_logic
);
end entity;

architecture main of PWM_G is
    signal cnt: unsigned(bits-1 downto 0) := (others=>'0');
begin
    process (Clk) is
    begin
        if (rising_edge(Clk)) then
            cnt <= cnt+1;
        end if;
    end process;
    Q <= '1' when D>cnt else '0';
end architecture;

architecture best of PWM_G is
    constant maxcount:unsigned(bits-1 downto 0) :=
(0=>'0',others=>'1');
    signal cnt: unsigned(bits-1 downto 0) := (others=>'0');
begin
    process (Clk) is
    begin
        if (rising_edge(Clk)) then
            if cnt < maxcount then
                cnt <= cnt+1;
            else cnt<= (others=>'0');
            end if;
        end if;
    end process;
    Q <= '1' when D>cnt else '0';
end architecture;

```

### *Ovládání LED pomocí PWM*

---

Vytvoříme si zapojení blink2, které bude podobné předchozímu, ale upravené. Použijeme čtyřbitový PWM. Mnohabitový dělič hodinového signálu 50 MHz zůstane, z něj si vygenerujeme signál „lfo“ (neboli „nízkofrekvenční oscilátor“), a tímto signálem budeme budit čítač, který bude postupně čitat 0..15. Tento signál pak budeme posílat do PWM k převodu na „analogovou hodnotu“. Jako hodiny pro PWM můžeme použít libovolný kmitočet, vyšší než „lfo \* 16“ a vyšší než výše zmíněných 6,1 kHz. Zde se

přímo nabízí poslat tam rovnou hodinový signál 50 MHz, ona to LED zvládne...

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

entity blink2 is
  port (
    clk: in std_logic;
    led, led3, led9: out std_logic
  );
end entity;

architecture qd of blink2 is

signal counter: std_logic_vector (25 downto 0):=
(others=>'0');
signal v: unsigned (3 downto 0):= (others=>'0');
signal lfo: std_logic;

begin

process (clk) is
begin
  if (rising_edge(clk)) then
    counter <= std_logic_vector(unsigned(counter) + 1);
    lfo <= counter(22);
  end if;
end process;

process (lfo) is
begin
  if (rising_edge(lfo)) then
    v <= v + 1;
  end if;
end process;

--led3<='1';
--led9<='1';
led <='1';

P: entity work.PWM(main)  port map (Q=>led3, Clk =>clk,
D=>v);
Pb: entity work.PWM(best)  port map (Q=>led9, Clk =>clk,
D=>v);
```

```
end;
```

Všimněte si, že tentokrát obsahuje zapojení všechny tři LED, které na [kitu](#) jsou. Použiju obě architektury, jak kanonickou, tak vylepšenou s plněním do 100 %, jednu připojím na led3 (pin číslo 3), druhou na led9 (pin číslo 9), prostřední LED (pin 7) nastavím do 1 (a protože jsou LED zapojené na log. 1, znamená to, že bude zhasnutá).

V zapojení dělím 50 MHz mnohabitovým čítačem „counter“, z jeho 23. bitu odebírám signál „lfo“. Ve druhém procesu počítám pulsy na signálu „lfo“ a měním podle nich hodnotu „v“ (0 – 15). Vytvářím si dvě instance entity PWM. Všimněte si, že jsem nezadával do kódu definici komponenty, místo toho vytvářím instance přímo přes „entity work.PWM“ a vybírám architekturu.

Až budete testovat, neudělejte stejnou chybu jako já: Udělal jsem si nový projekt, zapomněl jsem, že jsem si nenastavil přiřazení signálu pinům, a pak jsem se velmi divil, že zapojení nefunguje!

A protože jsou, jak jsem už zmínil, LED připojeny na log. 1, bude se zapojení chovat obráceně, tj. místo rozsvícení bude pohasínat. Nejjednodušší změna je nastavit  $v \leq v - 1$ ;

## Sigma-Delta

---

Druhá metoda digitálně-analogového převodníku, nazvaná [sigma-delta](#), pracuje na jiném principu: snaží se nalézt stejný poměr počtu 1 a 0, jaký odpovídá poměru ( $D / 2N$ ), kde  $N$  je šířka převodníku v bitech. Vnější rozhraní zůstává stejné jako u PWM, tedy hodiny, vstupní vektor a výstupní jednobitový signál. Výstupem sigma-delta převodníku je signál, který obsahuje pulsy nestejně délky, ale ve výsledném součtu odpovídá poměr času v log. 1 ku času v log. 0 požadovanému poměru. Algoritmus je jednoduchý: K čítači o dané šířce  $N$  se přičítá hodnota  $D$ . Pokud došlo k přenosu, je na výstup poslána log. 1 a od čítače se odečte hodnota  $2N$ , jinak je na výstupu log. 0.

Ukázkový čtyřbitový sigma-delta převodník jsem napsal takto:

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

```

use ieee.numeric_std.all;

entity SDM is
  port (
    Q: out std_logic;
    D: in unsigned(3 downto 0);
    Clk: in std_logic
  );
end entity;

architecture main of SDM is
  signal accumulator: unsigned (4 downto 0):=(others=>'0');
begin
  process (Clk, D) is
  begin
    if (rising_edge(Clk)) then
      accumulator <= ('0' & accumulator(3 downto 0)) + ('0'
& D);
    end if;
  end process;
  Q <= accumulator(4);
end architecture;

```

Zase platí, že pro hodnotu „1111“ není výsledek ideálních 100 %, ale jednou za 16 cyklů je nastavena 0.

Nevýhodou PWM i sigma-delta je to, že signál obsahuje parazitní „nosnou“ frekvenci, a pro další použití je třeba za výstup zapojit dolní propust, která tuto frekvenci odfiltruje. Sigma-delta modulace má ve výsledném signálu tuto parazitní nosnou ale mnohem vyšší a s různými frekvencemi, které se skládají do vysokofrekvenčního šumu, což může být někdy výhodnější (např. vyšší odstup těchto frekvencí od signálu, takže se lépe odfiltrovává).

Když si zkuste přidat další převodník pro třetí LED, můžete vidět, jaký je rozdíl mezi PWM a sigma-delta. Já okem pozoruju drobný rozdíl v oblasti nízkých hodnot – zdá se mi, že sigma-delta dokáže líp *prokreslit* nízkou intenzitu světla.

Převedení SDM na generickou podobu opět nechám na vás. Na závěr tradiční testbench:

```

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

```

```

entity test is
end entity;

architecture bench of test is
signal clk: std_logic:='0';
signal led3, led, led9:std_logic;

signal D: unsigned (3 downto 0);

begin
process -- generovani hodin
begin
  wait for 5 ps;
  clk <= not clk;
end process;

D <= "1000",
      "0111" after 400ps,
      "0000" after 800ps,
      "1111" after 1200ps;

P: entity work.PWM(best) port map (Q=>led, Clk =>clk, D=>D);
S: entity work.SDM port map (Q=>led3, Clk =>clk, D=>D);

end architecture;

```

## Ach, ta paměť...

Klopné obvody od A do Z... vlastně „od D do T“... a k tomu taky něco o tom, jak si ve VHDL vytvořit paměť RAM i ROM.

Klopné obvody snad není potřeba představovat nikomu, kdo se v elektronice dostal za kapitolu 3 – Základní hradla. Dokonce i ve stavebnici Logitronik 01 se čtyřmi NAND byl klopný obvod R-S už někde v půlce návodu. I já jsem tu už některé typy [představil](#), včetně [obvodu D](#). Pojďme ale systematictěji:

## Klopné obvody

---

### R-S

Nejjednodušší klopny obvod s dvěma asynchronními vstupy R a S a výstupem Q (u všech klopnych obvodů bývá k dispozici i negovaný výstup /Q, ale tady ho, jak se říká, *zanebdám*). Poskládáte si ho ze dvou hradel NAND či NOR (anebo ze dvou tranzistorů...), kde výstup jednoho je zaveden jako vstup druhého a vice versa. Obecně platí, že log. 1 na vstupu R překlopí obvod do stavu 0, log. 1 na vstupu S překlopí obvod do stavu 1 a pokud jsou oba vstupy v log. 0, obvod zůstává v tom stavu, v jakém byl předtím. 1 na obou vstupech uvede obvod do nedefinovaného stavu.

```
architecture behavioral of rsko is
begin
    process (R,S) is
        variable state: std_logic :='X';
        begin
            if (R='1' and S='0') then
                state:='0';
            elsif (R='0' and S='1') then
                state:='1';
            elsif (R='1' and S='1') then
                state:='X';
            end if;

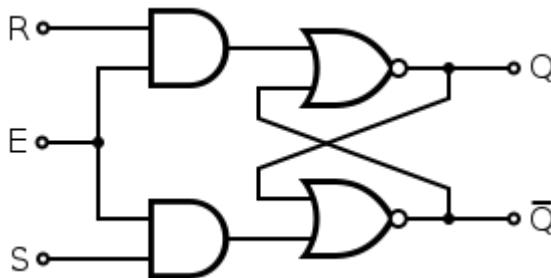
            Q <= state;
            nQ <= NOT state;

        end process;
end architecture;
```

---

### R-S s povolovacím vstupem

K předchozímu typu přidává další vstup E (Enable). Když je tento vstup v log. 0, jsou vstupy R a S „odpojeny“ a jejich změna se na stavu obvodu nijak neprojeví. Pokud je E v log. 1, obvod funguje jako výše uvedený.



Kód je jen mírně upravený předchozí:

```

architecture behavioral of rseko is

begin
    process (R,S,E) is
        variable state: std_logic :='X';
        begin
            if (R='1' and S='0' and E='1') then
                state:='0';
            elsif (R='0' and S='1' and E='1') then
                state:='1';
            elsif (R='1' and S='1' and E='1') then
                state:='X';
            end if;

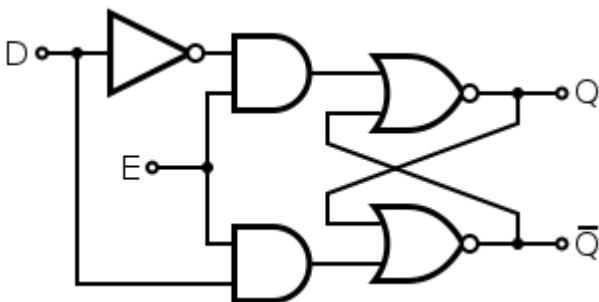
            Q <= state;
            nQ <= NOT state;

        end process;
    end architecture;

```

## D

Hazardní stav, kdy  $R = S = 1$ , můžeme eliminovat pomocí sloučení vstupů  $R$  a  $S$  u předchozího typu klopného obvodu do jednoho vstupu  $D$  (Data), který je připojen přímo na  $S$  a přes invertor na  $R$ . Tím zajistíme, že vstupy  $R$ ,  $S$  budou vždy v jednom stavu (0,1) nebo (1,0). Pomocí  $E$  pak určujeme, jestli se zapisuje nový stav (1), nebo jestli si obvod pamatuje předchozí (0). Výsledný klopný obvod nazýváme „D“ (v anglické literatuře DFF z „D Flip Flop“). Vstup  $E$  bývá označen jako  $C$  (Clock).



```

entity dff is
  port (
    D, C: in std_logic;
    Q: out std_logic
  );
end entity;

architecture main of dff is
begin
  process (C)
  begin
    if (rising_edge(C)) then
      Q <= D;
    end if;
  end process;
end architecture;

```

### *D s asynchronními vstupy*

Existuje „mutace“ výše uvedeného obvodu, kdy k obvodu D, který je synchronní (tj. řízený hodinovým vstupem) přidáme asynchronní vstupy R, S – tedy takové, které nulují či nastavují obvod bez ohledu na stav hodinového vstupu C.

```

entity dffrs is
  port (
    D, C, R, S: in std_logic;
    Q: out std_logic
  );
end entity;

architecture main of dffrs is
begin
  process (C, R, S)

```

```

begin
  if (R = '1') then
    Q <= '0';
  elsif (S = '1') then
    Q <= '1';
  elsif (rising_edge(C)) then
    Q <= D;
  end if;
end process;
end architecture;

```

### *Dělička*

Pokud na vstup D připojíme negovaný výstup Q, získáme na výstupu výstup s frekvencí, která odpovídá poloviční frekvenci, přivedené na vstup C.

### *T*

Klopný obvod T vznikne podobně jako předchozí dělička tím, že zavedeme negovaný výstup zpět na vstup D, tentokrát ale přes „povolovací“ hradlo AND ( $D \leq \text{not } Q \text{ AND } T$ ). Pokud je  $T=0$ , pamatuje si klopný obvod poslední stav, pokud je 1, tak se s každým pulsem hodin překlopí.

```

architecture main of tff is
  signal temp: std_logic := '0';
begin
  process (C)
  begin
    if (rising_edge(C)) then
      temp <= T xor temp;
    end if;
  end process;
  Q <= temp;
end architecture;

```

I k tomuto klopnému obvodu lze připojit synchronní či asynchronní vstupy pro nastavení / nulování.

### *Paměť*

Paměť jako elektronický prvek bývá implementována pomocí klopných obvodů (statická RAM či registry). Ve VHDL není potřeba vytvářet paměť takto složitě, stačí použít prosté:

```
type ram_t is array (0 to 255) of std_logic_vector(7 downto 0);
signal ram : ram_t := (others => (others => '0'));
```

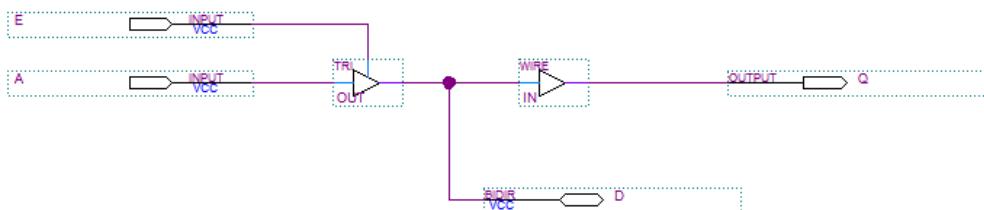
Nejprve deklarujeme typ ram\_t, který představuje 256bytovou paměť RAM („pole osmibitových hodnot“), no a na druhém řádku vytvoříme jeho fyzickou reprezentaci – signál „ram“ bude mít hodně daleko do běžné představy „signálu“ coby datového vodiče, ale nebojte, za chvíliku si ho „schováme“ do komponenty s běžnějším rozhraním. Všimněte si konstrukce s „others“. Zmiňoval jsem se, že tato konstrukce nastaví „všechny ostatní hodnoty, nezadané explicitně“ na určitou hodnotu. Tady nastavuje 256 položek na hodnotu „8 nul“.

V elektronice rozlišujeme dva základní typy pamětí: ROM (Read Only Memory) a RAM (Random Access Memory, přesněji: RWM – Read/Write Memory). Paměť ROM má vstupní adresovou sběrnici, výstupní datovou a vybavovací vstup (CE – Chip Enable, OE – Output Enable apod.) Paměť RAM má rovněž vstupní adresovou sběrnici, vstupní a výstupní datovou (někdy spojenou do obousměrné), vybavovací vstup a vstupní signál pro zápis hodnoty.

### Obousměrná sběrnice

Ve VHDL můžeme relativně snadno implementovat obousměrnou třístanovovou sběrnici. Při výběru směru v části PORT zadáme jako typ „INOUT“ – signál se od té chvíle chová jako vstup (tedy lze jej přiřadit jiným signálům), i jako výstup (tedy lze jemu přiřadit hodnotu).

Představme si jednobitový obousměrný vstup / výstup D. Pokud je řídicí signál E roven 1, vystupuje na tento výstup signál A. Pokud je řídicí signál E ve stavu 0, je signál D nastaven jako vstupní a jeho hodnota se propisuje do signálu Q. Nějak takto:



Kód pro takový obvod bude jednoduchý – využijeme možnosti přiřadit výstupu hodnotu „Z“ (vysoká impedance)

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

entity tris is
  port (
    A, E: in std_logic;
    Q: out std_logic;
    D: inout std_logic
  );
end entity;

architecture main of tris is
begin
  Q <= D;
  D <= A when E='1' else 'Z';
end architecture;
```

### Paměti RAM (RWM)

RAM si můžeme představit jako blok klopných obvodů typu D. Adresová sběrnice je zapojena na dekodér 1-na-N a každý výstup ovládá jeden klopný obvod. Pokud je dán požadavek zápisu, jsou vstupní data přivedena na vstup D daného klopného obvodu a jsou zapsána pulsem na hodinovém vstupu. Čtení dat probíhá obdobně – z výstupů všech klopných obvodů je multiplexorem vybrán požadovaný údaj (podle adresy).

Paměť RAM může být ve VHDL jednoportová či dvouportová (částečně nebo plně). Jednoportová RAM má jednu adresovou sběrnici a jeden vstup, určující, jestli se bude číst, nebo zapisovat. Může mít oddělenou vstupní a výstupní datovou sběrnici, nebo ji může mít obousměrnou. Částečně dvouportová paměť má adresovou + datovou sběrnici pro zápis a samostatnou adresovou + datovou sběrnici pro čtení. Plně dvouportová paměť má dvě nezávislé sady kompletních vývodů (data, adresa, řídicí signály), a znamená to, že v jeden okamžik mohou přistupovat dva různé obvody k téže paměťové matici s různými požadavky (zápis, čtení, z různých adres, nebo i ze stejných – zde pozor, při konkurenčním zápisu na stejnou adresu není výsledek zaručený, pokud nemá paměť definované priority vstupů).

V obvodech FPGA bývají speciálně vyhrazené bloky pamětí – právě do nich bývají alokována velká bitová pole. Jejich počet, velikost a organizace záleží na výrobci a typu. Pro zajímavost si popíšeme, jak je implementována paměť v obvodech Cyclone II (použitý v doporučeném začátečnickém kitu).

Cyclone II obsahují bloky paměti, nazývané M4K (Memory 4K), což je dvouportová paměť s velikostí 4608 bitů včetně paritních. Každý takový blok je možno organizovat do bloku  $4K \times 1$ bit,  $2K \times 2$ ,  $1K \times 4$ ,  $512 \times 8$ ,  $512 \times 9$ ,  $256 \times 16$ ,  $256 \times 18$ ,  $128 \times 32$  nebo  $128 \times 36$  bitů. Máme tedy půl kilobyte paměti.

Různé obvody z řady Cyclone II obsahují různé množství M4K bloků:

Typ	Počet bloků	Kapacita (bity)	Kapacita (kB)
EP2C5	26	119808	13
EP2C8	36	165888	18
EP2C15	52	239616	26
EP2C20	52	239616	26
EP2C35	105	483840	52,5

Typ	Počet bloků	Kapacita (bity)	Kapacita (kB)
EP2C50	129	594432	64,5
EP2C70	250	1152000	125

*Kapacita v kB se bere při organizaci po osmi bitech.*

**Cyclone IV** obsahují bloky paměti, nazývané M9K (Memory 9K), což je dvouportová paměť s velikostí 9216 bitů včetně paritních. Každý takový blok je možno organizovat do bloku 8Kx1bit, 4Kx2, 2Kx4, 1Kx8, 1Kx9, 512×16, 512×18, 256×32 nebo 256×36 bitů. Máme tedy jeden kilobyte paměti.

Různé obvody z řady Cyclone II obsahují různé množství M9K bloků:

Typ	Počet bloků	Kapacita (Kbits)	Kapacita (kB)
EP4CE6	30	270	30
EP4CE10	46	414	46
EP4CE15	56	504	56

Typ	Počet bloků	Kapacita (Kbits)	Kapacita (kB)
EP4CE22	66	594	66
EP4CE30	66	594	66
EP4CE40	126	1134	126
EP4CE55	260	2340	260
EP4CE75	305	2745	305
EP4CE115	432	3888	432

*Kapacita v kB se bere při organizaci po osmi bitech.*

Kromě vyhrazených bloků paměti (proto též „bloková paměť“) lze ve FPGA vytvořit tzv. „distribuovanou paměť“. Každá základní stavební buňka FPGA obsahuje několik klopných obvodů, které mohou sloužit k zapamatování dat, a syntetizér VHDL dokáže tyto obvody použít pro vytvoření paměti (ovšem takto použité buňky nelze už použít pro jiný účel). Distribuovaná paměť se proto hodí pro malé paměti. Bloková paměť je vhodnější pro větší paměti, jen je třeba počítat s její granularitou, tj. přiděluje se vždy po blocích. Jinými slovy – jedna stobytová paměť zabere jeden blok, sto jednobytových zabere sto bloků.

To, jestli se má vektor dat implementovat do blokové, nebo do distribuované paměti, rozhoduje syntetizér – většinou podle velikosti bloku dat a podle dalších indicií, např. zda je čtení i zápis synchronní. Ukažme si půlkilobyтовou paměť RAM (8 bitů) s oddělenými vstupními a výstupními daty a se dvěma řídicími signály – CE (Chip Enable, operace probíhají při náběžné hraně CE) a WE (1 = zapisuje se, 0 = nezapisuje se).

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

entity memo is
    port (
        Address: in unsigned(8 downto 0);
        Data_In: in std_logic_vector(7 downto 0);
        Data_Out: out std_logic_vector(7 downto 0);
        CE: in std_logic;
        WE: in std_logic
    );
end entity;

architecture memo of memo is
    type ram_t is array (0 to 511) of std_logic_vector(7
downto 0);
    signal ram : ram_t := (others => (others => '0'));

    -- Verze pro Alteru, data uložena v distribuované
paměti
    attribute ramstyle: string;
    attribute ramstyle of ram : signal is "logic";

    -- Verze pro Alteru, data uložena v blokové paměti
    -- attribute ramstyle: string;
    -- attribute ramstyle of ram : signal is "M4K";

    -- Verze pro Xilinx, data uložena v distribuované
paměti
    -- attribute ram_style: string;
    -- attribute ram_style of ram : signal is "distributed";

    -- Verze pro Xilinx, data uložena v blokové paměti
    -- attribute ram_style: string;
    -- attribute ram_style of ram : signal is "block";
```

```

begin

process(CE)
begin
    if (rising_edge(CE)) then
        Data_Out <= ram(to_integer(Address));
        if WE='1' then
            ram(to_integer(Address)) <= Data_In;
        end if;
    end if;
end process;

end architecture;

```

Na kódu není nic záludného nebo neznámého, s výjimkou atributu ramstyle (pro Xilinx ram\_style). Pomocí tohoto atributu můžeme explicitně určit, do jaké paměti se bude daný signál umisťovat. U mého testovacího kódu se vybere automaticky paměť M4K a výsledek je:

Total logic elements	0
Total combinational functions	0
Dedicated logic registers	0
Total registers	0
Total pins	27
Total virtual pins	0
Total memory bits	4,096
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Pro zajímavost, pokud vynutím umístění 512B paměti do distribuované (v názvosloví Altery to je „logic“), výsledek se radikálně změní (kromě toho, že samotné zpracování bude mnohonásobně delší):

Total logic elements	7,368
Total combinational functions	3,272
Dedicated logic registers	4,104
Total registers	4104
Total pins	27
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Vidíte, že tentokrát prosté pole 512 položek po 8 bitech, tedy 4096 bitů, vedlo k obsazení více než sedmi tisíc logických elementů, z toho 4104 paměťových a přes 3000 kombinačních. Proto vždy dbejte, aby se velká pole mohla umisťovat do blokové paměti, která je k podobným věcem určena.

**Pozor!** I když použijete správný atribut, tak se může stát, že váš vektor bude umístěn do distribuované paměti. Snadno taková situace vznikne, když použijete asynchronní čtení. Kdybychom ve výše uvedeném kódu přenesli řádek

```
Data_Out <= ram(to_integer(Address));
```

mimo synchronní část (tedy tu, která je ovládána vzestupnou hranou signálu CE), detekuje jej syntetizér jako možné „čtení během zápisu“, a takový přístup implementuje v distribuované paměti, protože bloková jej neumožňuje.

Zlatá pravidla tedy zní: Malé vektory klidně v distribuované, velké se snažte umístit do blokové; přečtěte si dokumentaci ke konkrétnímu obvodu, abyste věděli, jak se paměť nazývá a jaké má možnosti; do blokové paměti zásadně synchronně.

Pro zajímavost – varianta s obousměrnou datovou sběrnicí:

```
entity memo is
  port (
    Address: in unsigned(8 downto 0);
    Data: inout std_logic_vector(7 downto 0);
    CE: in std_logic;
    WE: in std_logic
  );
end entity;

architecture memo of memo is
  type ram_t is array (0 to 511) of std_logic_vector(7
downto 0);
  signal ram : ram_t := (others => (others => '0'));

  attribute ramstyle: string;
  attribute ramstyle of ram : signal is "M4K";

begin
```

```

process(CE)
begin
    if (rising_edge(CE)) then
        Data <= ram(to_integer(Address));
        if WE='1' then
            Data <= (others=>'Z');
            ram(to_integer(Address)) <= Data;
        end if;
    end if;
end process;

end architecture;

```

### *Paměť ROM*

Pro nejrůznější dekódovací tabulky, složitou kombinatoriku nebo třeba mikrokód využijeme paměť ROM. Ve FPGA nemáme nic jako ROM k dispozici, veškerá paměť je RAM, a tak si funkcionalitu ROM simulujeme tím, že neaktivujeme možnost zápisu. Na druhou stranu to znamená, že data musíme zadat *nějak jinak*.

U malých bloků to není problém udělat prostým přiřazením ve zdrojovém kódu:

```

TYPE memory IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO
0);
CONSTANT myrom: memory := (
    0  => "00011111",
    4  => "00111100",
    8  => "00000101",
    9  => "01010101",
    10 => "10100000",
    15 => "11111111",
    others => "00000000");

```

Pokud je paměť větší, byl by takový zápis krajně nepraktický. V takovém případě použijeme možnost zadat data v externím souboru. Altera používá formát MIF (Memory Initialization File), Xilinx používá COE, ale vývojové nástroje obou výrobců dokážou zpracovat Intel HEX file. To je tedy pravděpodobně nevhodnější varianta... *Tedy – byla by, pokud by zrovna například Altera nevyžadovala HEX v prapodivném formátu „4 byty na řádek“... Naštěstí existuje řešení!*

## *Hotové komponenty*

---

Vývojové prostředí Quartus obsahuje knihovnu hotových komponent, uzpůsobených na míru konkrétním FPGA od Altery. Odbornou knihovnu nabízí i Xilinx a další výrobci. V této knihovně najdete spoustu „vysokoúrovňových“ obvodů, jako jsou např. PLL, standardní rozhraní (PCIe, Ethernet, DisplayPort, ...), aritmetické obvody (násobičky, děličky, sčítáčky) nebo právě paměti. U Altery můžeme tyto obvody využít tak, že je začleníme do architektury a vhodně nastavíme GENERIC MAP a PORT MAP.

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

library lpm;
use lpm.lpm_components.all;

entity memo is
port (
    Address: in unsigned(12 downto 0);
    Data: out std_logic_vector(7 downto 0);
    CE: in std_logic
);
end entity;

architecture rom of memo is
begin
xrom:lpm_rom
generic map (
    lpm_widthad => 13,
    lpm_outdata => "UNREGISTERED",
    lpm_address_control => "REGISTERED",
    lpm_file => "rom8kb.hex",
    lpm_width => 8
)
port map (
    inclock=>CE,
    address=>std_logic_vector(Address),
    q=>Data
);
end architecture;
```

Zde jiná šířka .HEX souboru nevadí, syntetizér si s ní poradí. Všimněte si použití knihovny lpm, vytvoření instance entity „lpm\_rom“ a nastavení základních parametrů (šířka adresové sběrnice, datové sběrnice, soubor pro inicializaci atd.) Výsledkem je správně přeložená paměť ROM, umístěná v blokové paměti a po startu FPGA inicializovaná obsahem příslušného HEX souboru (ten se stane součástí konfigurace, nahrávané do konfigurační FLASH).

Knihovna hotových komponent se v IDE Quartus II skrývá pod skromným označením Megafuctions. Nejjednodušší způsob, jak s nimi pracovat, je využít MegaWizard Plug-In Manager (menu Tools). Zde si vyberete požadovanou komponentu, nastavíte její vlastnosti a necháte vygenerovat. Používáte ji pak jako jakoukoli jinou komponentu (průvodce vám vygeneruje i vzorový kód). Obdobné nástroje existují i pro FPGA od Xilinx.

Možná trochu překvapivým závěrem této kapitoly je: **Pokud chcete použít standardní typ paměti, netvořte ji ručně, ale použijte tu, kterou nabízí vaše vývojové prostředí!**

## Automaty

Trocha teorie o konečných stavových automatech, jejich implementaci ve VHDL, a jako bonus opravdové Hello World, tentokrát ne jako blikající LEDka, ale pěkně, řádně, přes sériové rozhraní!

### Konečné automaty

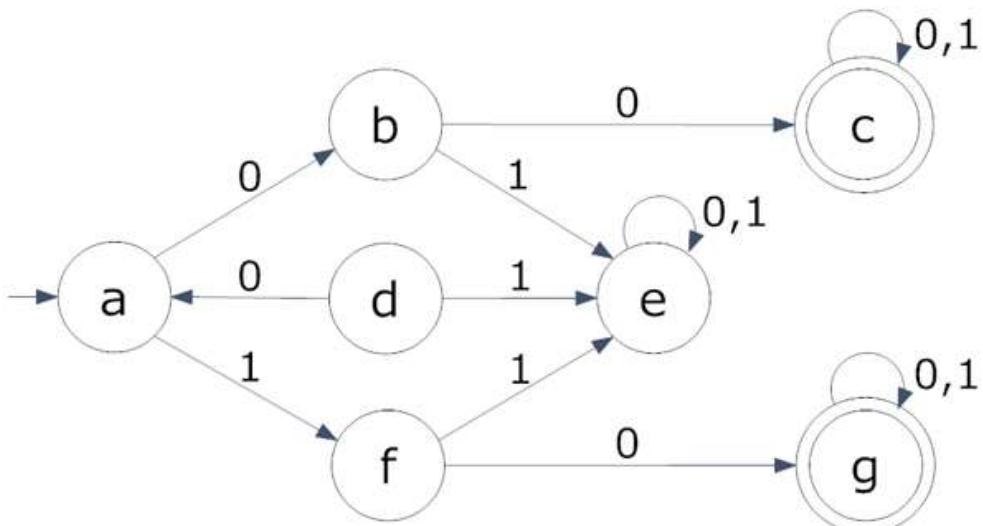
---

Už jsme si říkali, že ve VHDL není, jako v programování, nějaké „tohle, a pak tamto“ a že se věci dějí „najednou“. Samozřejmě s nadsázkou, ve skutečnosti mají obvody svá zpoždění a pokud navrhujete opravdu rychlé zapojení, tak je třeba s těmito časy počítat. Pro běžné „domácí“ použití ale nejsou tyto časy kritické, takže se můžeme na výsledek dívat tak, že se změny projevují „ihned“.

Jenže problém je, že často potřebujeme, aby se něco stalo v důsledku něčeho jiného, a poté se stalo ještě něco dalšího. Nejjednodušším případem je automat na bonbóny: čeká a nic nedělá. Jakmile někdo zmáčkne tlačítko, tak se dostane do stavu „počítej mince“. Když jich je dostatek, tak se

spustí proces „vydej bonbón“. Po něm následuje proces „vrať drobné“ a poté zase „čekej“. Když mincí není dostatek po nějakou dobu, nebo když člověk stiskne tlačítko „storno“, tak se jde na bod „vracení peněz“ a zpátky na čekání. Ve skutečnosti se taková logika dnes nejsnáze implementuje programovatelným obvodem (jednočipem, procesorem), ale princip je jasný.

Strojům, které jsou schopné mít nějaké různé stavы a na základě vnějších či vnitřních vlivů přecházet z jednoho do druhého říkáme **konečné stavové automaty**, anglicky FSM. Je k nim i celá [matematická teorie](#), kde se dělí na různé typy a druhy; souvisí například s regulárními výrazy. Teorii si můžete nastudovat tam, pro naše použití stačí barbarská definice: Konečný automat (FSM) je zařízení, které má schopnost zůstávat v různých stavech, a na základě vstupních signálů přecházet z jednoho stavu do druhého. Popisujeme jej množinou stavů, množinou možných vstupních signálů a **přechodovou funkcí**, která říká: Ze stavu X se v případě, že na vstupech je to a to, automat dostane do stavu Y. Kromě těchto tří vlastností je potřeba též určit, který stav je počáteční a které stavы jsou (případně) konečné. V elektronice bude takový „konečný stav“ většinou nějaká neopravitelná chyba, která vyžaduje fyzický restart.



Na obrázku je příklad automatu, který má sedm stavů (a až g) a jeden vstup. Uzly grafu udávají stavы, orientované hrany grafu říkají, za jakých podmínek se přechází z jednoho stavu do druhého. V tomto případě se začíná ve stavu „a“. Pokud je vstup „0“, přechází se do stavu „b“, pokud je vstup 1, přechází se do stavu „f“. A tak dál. Graf je redundantní, stavы „c“,

„e“ a „g“ jsou ekvivalentní a konečné a stav „d“ je nedosažitelný (nelze se dostat „do něj“).

Celý automat je řízen hodinovými pulsy – s příchodem hodinového pulsu se rozhoduje, jak se změní stav automatu.

Ve VHDL se řeší stavový automat například podle tohoto vzoru:

```
type states is (výčet stavů)
signal current_state, next_state: states;
---

process(clk, rst)  is begin
  if rst='1' then
    next_state <= {úvodní stav}
  elsif rising_edge(clk) then
    current_state <= next_state;
  end if
end process
---

process(current_state) is begin
  case current_state is
    when {stav1} =>
      -- nějaké operace pro tento stav
      -- případně nový stav se uloží do proměnné next_state
    when {stav2} =>
      -- nějaké operace pro tento stav
      -- případně nový stav se uloží do proměnné next_state
    -- atd.
    when others => null;
      -- tohle by nemělo nikdy nastat
  end case;
end process;

-- případné další věci, co se odehrávají v jednotlivých
stavech
```

Konkrétní implementace se může lišit – například se často spojuje synchronizační proces (tj. ten, který sleduje hodiny a při jejich příchodu nastaví správný aktuální stav) s přechodovou funkcí – její roli zde má proces s velkým „case“.

Konečné automaty použijeme ve spoustě nejrůznějších aplikací, od primitivních sekvenčních automatů (např. semafor na křižovatce) po implementaci mikroprocesoru.

## UART

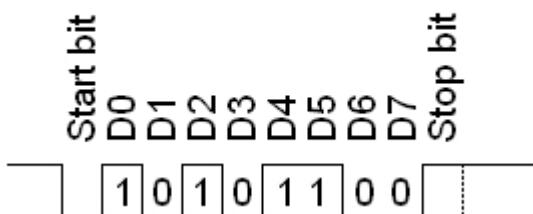
Dobře, uznávám, je to přehnané. Nebudeme implementovat celý UART (Univerzální asynchronní přijímač a vysílač), ale jen jednu jeho část, totiž vysílač, a nebudeme ji implementovat univerzálně, ale docela natvrdo. Univerzální rozšíření si můžete dodělat za domácí úkol.

Nejprve trocha teorie: Sériový vysílač bere osmibitová vstupní data, a jakmile dostane signál „vysílej!“, vyšle je na jednobitový výstup Tx. Tx je normálně v logické 1. Jakmile je dán požadavek na vysílání, je vyslán start bit („0“), pak jsou vyslány bity od nejnižšího po nejvyšší, po nich případně paritní bit (není vyžadován a já ho neimplementoval) a nakonec jeden, dva nebo „jeden a půl“ stop bitu („1“). Doba trvání jednotlivých pulsů je dána vysílací frekvencí a udává se v baudech = bitech za sekundu. Pozor, je potřeba si uvědomit, že osmibitové vysílání zabere nejméně 10 bitů (start bit + 8 bitů dat + 1 stop bit) a nepočítat maximální přenosovou rychlosť v bajtech jako „rychlosť / 8“!

U synchronního vysílání se spolu se signálem přenášejí i hodiny, u asynchronního je potřeba nastavit vysílač i přijímač na stejnou frekvenci. Používají se frekvence odvozené od frekvence 150 Hz, tedy 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 či 115200 Hz.

Proto se u sériových portů nastavuje mnoho parametrů, kromě hodin i počet stop bitů či parita, a k tomu další způsoby řízení přenosu (dtr, rts, dsr, cts). My si implementujeme přenos 9600/8-N-1, tedy 9600 Bd, 8 bitů, bez parity (N) a s jedním stop item, bez hardwarového řízení přenosu.

**Data is h'35' = b'00110101'**



Implementace vysílače je poměrně přímočará. Můžeme použít čítač, který je ve stavu 0, a jakmile přijde signál „Vysílej!“, tak začne čítat hodinové impulsy. Při hodnotě 1 se vyšle log. 0, při hodnotách 2 až 9 se budou vysílat jednotlivé bity, při hodnotě 10 se vyšle log. 1 a při hodnotě 11 se opět vynuluje. Ale pojďme použít FSM.

## *Struktura*

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

-- no parity, 1 stop bit

entity uart_tx is generic (
    fCLK : integer := 50_000_000;
    fBAUD : integer := 9600
);
port (
    clk, rst: in std_logic;
    tx: out std_logic:='1';

    tx_send: in std_logic; --send data
    tx_ready: out std_logic; -- transmitter ready
    tx_data: in std_logic_vector(7 downto 0)

);
end entity;
```

Entitu uděláme alespoň trošku generickou – generické parametry jsou rychlosť systémových hodin a vysílací rychlosť v baudech. Port tvoří hodinový vstup „clk“, nulovací vstup „rst“, sériový výstup tx, datový vstup tx\_data, řídicí vstup tx\_send a stavový výstup tx\_ready.

Vstupem „rst“ nulujeme celý obvod, vstupem tx\_send dáváme signál „Vysílej!“, signálem tx\_ready říká obvod, zda právě vysílá (0), nebo zda je připraven vysílat (1). Zbytek asi nepotřebuje vysvětlení.

## *Hodiny*

Ačkoli se to nezdá (a na začátku jsem psal, že se „věci dějí najednou“), tak je programovatelná logika založená na hodinovém signálu. Hodiny jsou „svaté“, bez nich máme jen poměrně tupou kombinační logiku. Někdy si probereme složitější téma, jakým jsou hodinové domény a přechod signálu mezi nimi, což je poměrně zásadní věc u složitých obvodů. Pro naše použití si teď vystačíme s jedním „centrálním časem“ (v případě [doporučeného začátečnického kitu](#) je to 50 MHz) a ostatní časy si odvodíme z něj. V našem případě potřebujeme získat hodinový signál o frekvenci 9600 Hz. Nejjednodušší postup je počítat pulsy hlavních hodin, a jakmile jich napočítáme N, tak hodíme puls na vysílací hodiny a vynulujeme čítač. N je číslo,

rovné podílu frekvence hodin a frekvence vysílání, tedy  $50\text{MHz} / 9600\text{ Hz} = 5208,333\dots$  Použijeme hodnotu 5208, výsledný kmitočet tedy bude  $9600,6144\dots$  což je v toleranci.

Hodinový dělič bude nulován vstupem rst.

```
architecture main of uart_tx is
    constant baudrate: integer:=(fCLK / fBAUD); --5208
    signal baudclk: std_logic;
    ... další deklarace ...
begin
    clock: process(clk)
        variable counter: integer range 0 to baudrate-1 :=0;
        begin
            if rising_edge(clk) then
                if counter = baudrate-1 then
                    baudclk <= '1';
                    counter := 0;
                else
                    baudclk <= '0';
                    counter := counter + 1;
                end if;
                if rst='1' then
                    baudclk <= '0';
                    counter := 0;
                end if;
            end if;
        end process;
    ...

```

Vidíme proměnnou counter, která počítá od 0 do baudrate-1. Baudrate je výše zmíněná konstanta, vzniklá podlením frekvencí fCLK a fBAUD. Counter je inicializován na hodnotu 0. S příchodem hodinového pulsu kontrolujeme, zda už máme načítáno dost (pak pouštíme puls na interní signál baudclk a nulujeme počítadlo), nebo zda počítáme dál. Pokud se objeví signál rst, nulujeme počítadlo i vnitřní hodiny.

### Automat

Náš automat bude mít tři stavů: idle, data a stop. Je velmi jednoduchý, lineární. Vstupní stav je idle. V tomto stavu je tx=1 (klidový stav), tx\_ready=1 (obvod je připraven vysílat) a čeká se na příchod signálu tx\_send. Jeho příchod způsobí hned několik věcí: hodnota z tx\_data se zkopiuje do interního bufferu, vynuluje se výstup tx\_ready, nastaví se počítadlo vy-

sílaných bitů na 7, spustí se vysílání start bitu ( $tx=0$ ) a nastaví se, že následující stav je data. Prodleva mezi přechodem stavového automatu vytvoří potřebně dlouhý start bit.

Ve stavu data se vysílají jednotlivé byty. Na výstup tx je zkopírován nejnižší bit z bufferu, buffer rotuje o 1 bit doprava a snižuje se počítadlo bitů. Pokud je nenulové, zůstáváme ve stavu data, jakmile je nulové, přecházíme do stavu stop.

Ve stavu stop nastavíme výstup tx na 1 (stop bit) a řekneme, že s příštím příchodem hodin se přechází do stavu idle.

Celý stavový automat je řízen vysílacími hodinami. Proces ale není postavený na sledování signálu baudclk, ale clk (máme jedny hodiny, které vládnou všem...). Takže s příchodem pulsu na clk se kontroluje, jestli náhodou není už i vhodná doba podle „baudclk“. Pokud ne, neděje se nic, pokud ano, spouští se další iterace stavového automatu.

V implementaci automatu jsem nepoužil dvě proměnné (aktuální stav a následující), ale pouze jednu. Přechodová funkce je tak jednoduchá, že to takto stačí.

Zavedl jsem ale interní signál tx\_en, který je „synchrozní náhradou“ signálu tx\_send. Totiž – při testech jsem narázela na problém, že obvod nechtěl vysílat. Po několika pokusech jsem zjistil, že problém je v tom, že signál tx\_send musí být aktivní minimálně po dobu jednoho pulsu vysílačích hodin (baudclk), a to jsem neměl (používal jsem krátké pulsy). Proto jsem si zavedl signál tx\_en, který je 0 a v případě, že přijde puls tx\_send, tak se nastaví na 1 a podrží tak informaci o zahájení vysílání až do okamžiku, kdy se zpracovávají vysílací hodiny. Vlastní stavový automat pracuje až s tímto signálem tx\_en. Výsledkem je, že pro spuštění vysílání stačí velmi krátký impuls tx\_send. Dalším důsledkem této změny je, že jsem přesunul kopírování dat do interního bufferu právě do tohoto bodu – bylo by podivné reagovat na krátký signál tx\_send, ale data si načíst až za „dlouhou dobu“.

```
architecture main of uart_tx is
-- deklarace hodin, viz výše
  type state is (idle, data, stop);
  signal fsm: state:=idle;

  signal data_temp: std_logic_vector(7 downto 0);
  signal datacount: unsigned(2 downto 0);
```

```

signal txen:std_logic:='0';

begin
    clock: process(clk)
        -- viz výše
    end process;

transmit: process(clk)
begin
    if rising_edge(clk) then
        if tx_send='1' and fsm=idle then
            txen <='1';
            data_temp<=tx_data;
        end if;

        if baudclk='1' then

            tx<='1';
            case fsm is

                when idle =>
                    tx_ready<='1';
                    if txen='1' then
                        datacount<=(others=>'1');
                        tx<='0'; --start bit
                        fsm <= data;
                        tx_ready<='0';
                        txen<='0';
                    end if;

                when data =>
                    tx<=data_temp(0);
                    tx_ready<='0';
                    if datacount=0 then
                        fsm<=stop;
                        datacount<=(others=>'1');
                    else
                        datacount<=datacount-1;
                        data_temp<='0' & data_temp(7
downto 1);
                    end if;

                when stop =>
                    tx<='1'; --stop bit
                    txen<='0';
                    fsm<=idle;
            end case;
        end if;
    end if;
end process;

```

```

        tx_ready<='0';

        when others => null;
    end case;

    if rst='1' then
        fsm <= idle;
        tx<='1';
        txen<='0';
    end if;

    end if; --baudclk
end if; --rising_edge(clk)
end process;

end architecture;

```

Po „velkém CASE“ je ještě jedna část, která ošetřuje signál rst: nastaví automat do stavu idle, výstup do klidového stavu, nuluje vysílací signál tx\_en.

### Hello...

Takovou komponentu použiju ve velmi primitivním zapojení: jeden čítač postupně čítá od 0 do 7 a adresuje tak osmibajtovou paměť ROM, ve které je uložena zpráva. Když vysílač hlásí „ready“, přejde čítač na další adresu a pošle signál tx\_send. Výstup paměti ROM je zapojen na vstup tx\_data. Takže se posílá stále dokola osmiznakový vzkaz („Hello!\n\r“).

```

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

entity rshello is
port (clk:in std_logic;tx:out std_logic);
end entity;

architecture fsm of rshello is
type msg is array(0 to 7) of std_logic_vector(7 downto 0);
signal str: msg
:= (X"48",X"65",X"6C",X"6C",X"6F",X"21",X"0D",X"0A");
signal data:std_logic_vector(7 downto 0);
signal ready: std_logic;
signal send: std_logic :='1';

begin

```

```

process(ready, clk) is
    variable cnt: unsigned(2 downto 0):="111";
begin

    if rising_edge(clk) then
        if ready='1' then
            cnt := cnt + 1;
            data <= str(to_integer(cnt));
            send <= '1';
        else
            send<='0';
        end if;
    end if;

end process;

transmitter: entity work.uart_tx port map (clk, '0', tx,
send, ready, data);
end architecture;

```

Po vhodném nastavení hodinového vstupu a datového výstupu se můžeme ke kitu připojit přes převodník USB-UART a v obyčejném sériovém terminálu (9600/8-N-1) uvidíme vzkaz našeho FPGA světu.

Tentokrát už doopravdy.

## Hodinové domény

Oblast, které se nelze vyhnout, pokud má vaše zařízení komunikovat se světem.

Už v předchozím textu jsem zmiňoval, že hodiny jsou svaté. Je to lehká nadsázka, ale rád bych teď probral podrobněji, co jsem tím myslел.

V odborné literatuře se o hodinových doménách můžete dočíst spoustu teorie. Je to užitečné vědět a pokud to myslíte s návrhem obvodů vážně, musíte tuto problematiku dobře znát. Doporučuju tedy nastudovat, ale protože začátečníka může silně technický popis problému vyděsit, nabízím takový stručný úvod do problematiky.

## Hodinové domény

---

Ve světě elektronických obvodů je potřeba synchronizovat činnosti. S kombinační logikou si člověk nevystačí, a pokud zavede asynchronní kombinační zpětnou vazbu do obvodu (viz například [klopný obvod R-S](#)), může se stát, že narazí na hazardní stavy, kdy se třeba obvod rozkmitá na frekvenci, omezené jen zpožděním signálů v logických členech. Proto se pro jakékoli složitější zapojení používá synchronizace pomocí hodinových pulsů. Hodiny mívají takovou frekvenci, aby zpoždění a doby náběhu, předstíhy a přesahy jednotlivých komponent neměly vliv na funkci celého obvodu. Obvody se navrhují tak, že zpravidla reagují všechny na stejnou (např. náběžnou) hranu hodinového signálu. Ta pak udává „takt“, v jakém celé zapojení pracuje.

Ideální je, pokud si celé zapojení vystačí s jedněmi hodinami. Takhle například fungoval náš [blikáč](#). Někdy je potřeba mít více frekvencí, což byl případ [sériového vysílače v minulé kapitole](#). Tam ale naštěstí šla nižší frekvence odvodit od té vyšší, prostým podělením. Oba hodinové signály tak měly stejnou fázi (tj. náběžná hrana toho pomalejšího přicházela s náběžnou hranou toho rychlejšího).

Někdy to ale není možné. Někdy přichází signál, který má vlastní časování, vlastní takt. Jeho hodiny mohou mít stejnou frekvenci, ale pravděpodobně budou mít nejen jinou, ale i nesoudělnou. A i když budou mít frekvenci soudělnou (např. poloviční, třetinovou, desetinovou), tak se pravděpodobně bude lišit fáze (tj. náběžné hrany nepřicházejí ve stejný okamžik).

Problém nastává, když z obvodu s jedněmi hodinami přechází signál do obvodu s jinými. Vlivem různých hodinových frekvencí (analogie: *sampling frekvence*) se může stát leccos. Krátké impulsy nemusí být zachyceny a mohou se ztratit. Změna hodnoty nemusí být zaznamenána. Impulsy se vlivem fázových rozdílů neúnosně zkrátí. A tak dál.

Další problém přináší *metastabilita klopných obvodů*. Pokud je klopný obvod citlivý např. na vzestupnou hranu hodinového signálu, je potřeba, aby datový vstup byl ustálený chvíli před příchodem této hrany (aby měl dostatečný *předstih*) a aby zůstal ustálený i chvíli po příchodu této hrany (aby měl dostatečný *přesah*). Zároveň je potřeba nechat obvodu určitý čas na zotavení po resetu. Všechny tyto časy jsou velmi krátké, ale v případě lehce fázově posunutých hodin se může signál právě do těchto bodů strefit. Po-

kud se tak stane, může se na výstupu klopného obvodu objevit nejednoznačný signál. Může přijít krátký parazitní zákmit, výstup se může rozkmitat, popřípadě být v neurčitém stavu...

### *Jak se vyhnout problémům?*

Je několik způsobů, jak se vyhnout problémům. Úplně ten nejjednodušší je: Mít všude jen jeden základní hodinový signál.

Jenže to ne vždy jde. Jakmile připojíte obvod do „vnějšího světa“, začnou chodit různé asynchronní vstupy. Například sériový přenos po sběrnici SPI, ten si s sebou nese vlastní hodinový signál, nebo klávesnice s rozhraním PS/2, to jsou nejkřiklavější příklady rozdílných hodinových domén.

Je proto potřeba každý signál ošetřit. Ošetření se liší podle podstaty toho kterého signálu.

### *Průchod pomalých signálů*

U dlouhých impulsů je potřeba zajistit, aby prošly náběžné i sestupné hrany, které jsou synchronizovány se vstupními hodinami CLKin, a aby vznikly ekvivalentní náběžné a sestupné hrany, synchronizované s cílovými hodinami CLKout. Pokud CLKin < CLKout, nebývá to zas tak velký problém – cílový systém běží na vyšší frekvenci a je schopen hrany přebírat s minimálním zpožděním. U opačného případu se impulzy mohou prodloužit či zkrátit, a příliš krátké impulsy rychle za sebou nemusí projít.

Nejjednodušší by bylo připojit signálu do cesty klopný obvod D s hodinami připojenými na CLKout, tedy na hodiny cílového obvodu. Vzhledem k možné metastabilitě takového zapojení (signál se může změnit nebezpečně blízko hodinového pulsu) zapojujeme dva obvody za sebou. Ve VHDL pak zapisujeme například pomocí dvoubitového „posuvného registru“.

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

entity sync0 is
port (
    CLKout: in std_logic;
    Din: in std_logic;
    Dout: out std_logic
);
end entity;
```

```

architecture main of sync0 is
signal sync:std_logic_vector(1 downto 0);
begin

    process (CLKout) is
    begin
        if rising_edge(CLKout) then
            sync(1) <= sync(0);
            sync(0) <= Din;
        end if;
    end process;

    Dout <= sync(1);

end architecture;

```

U pomalých signálů, což jsou v našem případě signály delší než trvání hodinového cyklu cílové oblasti, můžeme ignorovat zdrojové hodiny a používat jen ty cílové. V procesu máme dvoubitový signál „sync“, kde vstup dat jde do sync(0), sync(0) se posouvá do sync(1) a sync(1) tvoří výstup dat. Takto jsou vytvořeny dva klopné obvody D. Nevýhoda je, že se v signálu objeví zpoždění.

Pro testování synchronizace jsem si připravil testbench, ve kterém jsou tři různé hodinové signály: rychlý clk1 s periodou 10ns, pomalý clk2 s periodou 103ns (mírným „rozladěním“ proti celočíselnému násobku dosahují fázového posunu) a opět rychlý clk3 s periodou 9ns (řádově stejně rychlý jako clk1, ale s drobným rozladěním, které vede k fázovým posunům).



Na obrázku je v detailu vidět, jak se hodiny 1 a 3 posunou o  $180^\circ$  a jak hodiny 2 mají hranu úplně mimo.

Do série zapojuji dva synchronizační obvody. První je mezi clk1 a clk2, druhý mezi clk2 a clk3. Signál tak prochází z rychlé hodinové domény do pomalé, a z ní opět do rychlé. Generuju dva signály, slow a strobe, synchronizované s hodinami clk1.

```

architecture bench of test is
signal clk1, clk2, clk3: std_logic:='0';
signal din, dmld, dout: std_logic:='0';

```

```

signal slow, strobe: std_logic:='0';

begin

    c1: process
    begin
        wait for 10ns;
        clk1 <= not clk1;
    end process;

    c2: process
    begin
        wait for 103ns;
        clk2 <= not clk2;
    end process;

    c3: process
    begin
        wait for 9ns;
        clk3 <= not clk3;
    end process;

--slow signal
slowsig: process (clk1)
variable cnt: integer range 0 to 31:=0;
begin
    if rising_edge(clk1) then
        if cnt=31 then
            slow <= not slow;
            cnt := 0;
        else
            cnt := cnt+1;
        end if;
    end if;
end process;

--strobe signal
strob сиг: process (clk1)
variable cnt: integer range 0 to 31:=0;
begin
if rising_edge(clk1) then
    if cnt=31 then
        strobe <= '1';
        cnt := 0;
    else
        strobe <= '0';
    end if;
end if;
end process;

```

```

        cnt := cnt+1;
    end if;
end if;
end process;

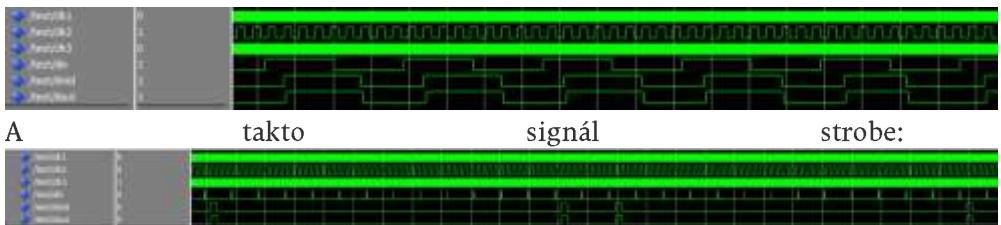
din<=strobe;

uut1: entity work.sync1 port map (clk1, clk2, din, dmid);
uut2: entity work.sync1 port map (clk2, clk3, dmid, dout);

end architecture;

```

Takto prochází obvodem signál slow (Din -> Dmid -> Dout):



Myslím, že nebudu přehánět, když napíšu, že máme vážný problém... Prošel zhruba každý patnáctý. Pro dlouhé signály je tento přístup použitelný, pro krátké rozhodně ne.

### Průchod krátkých pulsů

Pokud potřebujeme korektně zpracovat krátké pulsy, musíme použít složitější způsob. Uvnitř synchronizačního obvodu si udržujeme informaci o tom, že „přišel脉冲“ (samozřejmě řízenou hodinovým kmitočtem zdroje), a pokud přišel, tak na výstupu vytvoříme脉冲 podle hodin cílového obvodu. Například takto:

```

library ieee;
use ieee.std_logic_1164.all;

entity sync1 is
port (
    CLKin: in std_logic;
    CLKout: in std_logic;
    Din: in std_logic;
    Dout: out std_logic
);
end entity;

```

```

architecture main of sync1 is
signal sync:std_logic_vector(2 downto 0):="000";
signal flag:std_logic:='0';
begin

    process (CLKin) is
    begin
        if rising_edge(CLKin) then
            flag<=flag xor Din;
        end if;
    end process;

    process (CLKout) is
    begin
        if rising_edge(CLKout) then
            sync(2) <= sync(1);
            sync(1) <= sync(0);
            sync(0) <= flag;
        end if;
    end process;

    Dout <= sync(1) xor sync(2);

end architecture;

```

Takto napsaný obvod „propouští hrany“ s určitým zpožděním. Tentokrát je výsledek podstatně lepší:



Vidíme, že pomalejší část poctivě tvoří pulsy podle krátkých pulsů na vstupu, s určitým zpožděním, a při konverzi z pomalých hodin na rychlé vznikají opět krátké pulsy.

Někdy může být výhodné, když dá převodník zdvojovému obvodu vědět, že puls ještě nedorazil do cílového obvodu. Princip spočívá ve spojení obou předchozích technik do dvousměrného synchronizačního obvodu.

```

library ieee;
use ieee.std_logic_1164.all;

entity sync1a is
port (
    CLKin: in std_logic;

```

```

CLKout: in std_logic;
Din: in std_logic;
Dout: out std_logic;
Busy: out std_logic
);
end entity;

architecture main of sync1a is
signal sync:std_logic_vector(2 downto 0):="000";
signal syncB:std_logic_vector(1 downto 0):="00";
signal flag:std_logic:='0';
signal tBusy: std_logic;
begin

process (CLKin) is
begin
    if rising_edge(CLKin) then
        flag<=flag xor (Din and not tBusy);
    end if;
end process;

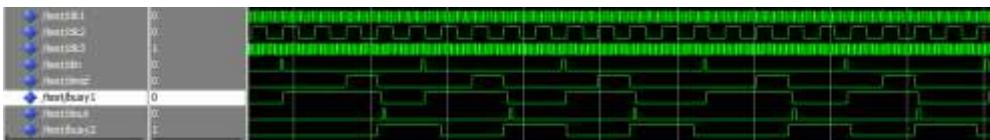
process (CLKin) is
begin
    if rising_edge(CLKin) then
        syncB(1) <= syncB(0);
        syncB(0) <= sync(2);
    end if;
end process;

process (CLKout) is
begin
    if rising_edge(CLKout) then
        sync(2) <= sync(1);
        sync(1) <= sync(0);
        sync(0) <= flag;
    end if;
end process;

Dout <= sync(1) xor sync(2);
Busy<=tBusy;
tBusy<=flag xor syncB(1);

end architecture;

```



### *Složitější informace*

Pomocí těchto základních principů můžeme vytvářet složité synchronizační obvody, které dokážou přenášet několik signálů s různým významem, například informace o zahájení zpracování údajů, o konci zpracování, ... Co když potřebujeme přenášet vícebitové hodnoty?

Pokud jde o prosté monotónní řady hodnot (čítání), pak zvažte převod do Grayova kódu. Grayův kód má tu výhodu, že při přechodu mezi hodnotami s krokem 1 se vždy mění pouze jeden jediný bit. To usnadňuje přechod mezi doménami, protože přichází vždy jen jedna hrana v jednu chvíli. Pokud bychom spolehlali na změnu více hran naráz, mohli bychom se opět ocitnout v hazardním stavu...

Pokud chceme přenášet obecně paralelní data, můžeme zvolit dva přístupy. Prvním je „převzorkování“ – na všechny bity použijeme výše uvedenou synchronizaci, a výsledek považujeme za směrodatný, i když počítáme s tím, že nám mohou některá data uniknout. Druhý přístup je přístup pomocí vyrovnávací paměti (bufferu), kam jedna doména zapisuje a druhá čte.

### *UART, druhý díl – přijímač*

V předchozím textu jsem nakousl implementaci vysílače sériových dat ve tvaru 8-N-1. Teď si ukážeme ideový protipól, tedy přijímač dat. Pokud jste se zamýšleli nad tím, jak takový přijímač implementovat, přišli jste na to, že problémem je synchronizace s přicházejícím signálem. Ten totiž ignoruje naše vnitřní hodiny a přijde si, kdy se mu zlídí. U vysílače to až tak nevadilo – asynchronní puls na vstupu tx\_send jsem si „pozdržel“ až do okamžiku, kdy přišel impuls na „vysílacích“ hodinách. V podstatě jsem také srovnával dvě hodinové domény...

U přijímače na to nemůžeme spoléhat. Tam přicházející sestupná hrana oznamuje stop bit, a je na přijímači, aby v tu chvíli spustil hodiny a podle této hrany přijímal data. Přístupy jsou různé. Někdo volí „oversampling“, tedy interní běh na vyšší harmonické frekvenci, např. 16x vyšší, než je

frekvence vysílání. Já jsem zvolil jiný přístup, kdy opravdu fyzicky držím hodiny vypnuté, s příchodem sestupné hrany je zapínám, a navíc první cyklus zkrátím o polovinu, abych se dostal vždy do středu předpokládaného intervalu a nevzorkoval v blízkosti hran. V ideálním případě tak čtu hodnotu vždy v polovině intervalu.

Do přijímače jsem implementoval i jednoduchý „low pass“ filtr pomocí posuvného registru. Krátké impulzy (rušení) tak nespustí případné přijímání dat. Filtr funguje tak, že do čtyřbitového posuvného registru vstupují vstupní data, a na výstupu je hodnota 0, pokud jsou všechny bity ve stavu „0000“, hodnota 1, pokud jsou všechny bity ve stavu „1111“, a pokud je kombinace jiná, tak se výstup nemění. Uvnitř zapojení už pracuju jen s tímto filtrovaným signálem.

Generátor hodin funguje podobně jako u vysílače, s tím rozdílem, že je řízen interním povolovacím signálem clken.

Detektor sestupné hrany pracuje tak, že si ukládá předchozí stav vstupu rx (filtrovaného) a porovnává ho s aktuálním stavem. Pokud je interní automat ve stavu „idle“, tak příchod sestupné hrany spustí hodiny (clken). Naopak návrat do idle a klid na lince znamená zastavení hodin.

Samotný přijímač je pak tvořen opět konečným automatem, který je obdobou toho z vysílače.

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

-- no parity, 1 stop bit

entity uart_rx is generic (
    fCLK : integer := 50_000_000;
    fBAUD : integer := 9600
);
port (
    clk, rst: in std_logic;
    rx: in std_logic:='1';

    rx_valid: out std_logic:='0'; -- data valid
    rx_data: out std_logic_vector(7 downto 0):=(others=>'0')
);
end entity;
```

```

architecture main of uart_rx is
    type state is (idle, start, data, stop);

    constant baudrate: integer:=(fCLK / fBAUD); --5208
    constant halfbaudrate: integer:=(baudrate / 2); --2604

    signal fsm: state:=idle;
    signal baudclk: std_logic;
    signal data_temp: std_logic_vector(7 downto
0):=(others=>'0');
    signal datacount: unsigned(2 downto 0):=(others=>'1');
    signal rxflt:std_logic:='1';
    signal clken:std_logic:='0';

begin

    filter: process(clk) is
        variable flt: std_logic_vector(3 downto 0);
    begin
        if rising_edge(clk) then
            if flt="0000" then
                rxflt<='0';
            elsif flt="1111" then
                rxflt<='1';
            end if;

            flt := flt(2 downto 0) & rx; -- flt <<< rx
        end if;
    end process;

    clock: process(clk)
        variable counter: integer range 0 to baudrate-1 :=0;
    begin
        if rising_edge(clk) then
            if counter = baudrate-1 then
                baudclk <= '1';
                counter := 0;
            else
                baudclk <= '0';
                counter := counter + 1;
            end if;
            if rst='1' then
                baudclk <= '0';
                counter := 0;
            end if;
            if clken='0' then
                baudclk <= '0';
            end if;
        end if;
    end process;

```

```

        counter := halfbaudrate;
    end if;
end if;
end process;

detect: process(clk) is
    variable old_rx:std_logic:='0';
begin
    if rising_edge(clk) then
        --detekce sestupné hrany
        --pokud předtím bylo 1 a teď je 0 a stav je
idle
        if old_rx='1' and rxflt='0' and fsm=idle then
            clken<='1'; --zasynchronizujeme hodiny.
První cyklus poloviční
        end if;
        if old_rx='1' and rxflt='1' and fsm=idle then
            clken<='0'; --vypneme hodiny.
        end if;
        old_rx := rxflt;

        if rst='1' then
            clken <= '0';
            old_rx := '0';
        end if;
    end if;
end process;

receive: process(clk)
begin
    if rising_edge(clk) then

        if baudclk='1' then
            case fsm is

                when idle =>

                    if rxflt='0' then
                        datacount<=(others=>'1');
                        fsm <= data;
                        rx_valid<='0';
                    end if;

                when data =>
                    data_temp<=rxflt & data_temp(7
downto 1);
                    if datacount=0 then

```

```

        fsm<=stop;
        datacount<=(others=>'1');
    else
        datacount<=datacount-1;
    end if;

        when stop =>
            fsm<=idle;
            rx_valid<='1';
            rx_data <= data_temp;

        when others => null;
    end case;

end if; --baudclk
if rst='1' then
    fsm <= idle;
    rx_valid<='0';
end if;

end if; --rising_edge(clk)
end process;

end architecture;

```

Přijímač posílá ven signál „rx\_valid“, který oznamuje, že byla přečtena platná data. Nijak neřeší případné přetečení bufferu (taky neřeší, zda si už klient data vyzvedl), to je potřeba případně ošetřit v jiných částech.

### *Testbench*

```

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

entity test is
end entity;

architecture bench of test is
signal clk: std_logic:='0';
signal rst,tx,send,ready: std_logic;
signal rxdata,data: std_logic_vector(7 downto 0);
signal rxready: std_logic;
begin

```

```

        data <= x"55", x"aa" after 7 us;

    uut: entity work.uart_tx generic map (fBAUD=>1000000)
port map (clk, rst, tx, send, ready, data);
    rec: entity work.uart_rx generic map (fBAUD=>1000000)
port map (clk, rst, tx, rxready, rxdata);

    main_clock_generation:process
begin
    wait for 10ns;
    clk           <= not clk;
end process;

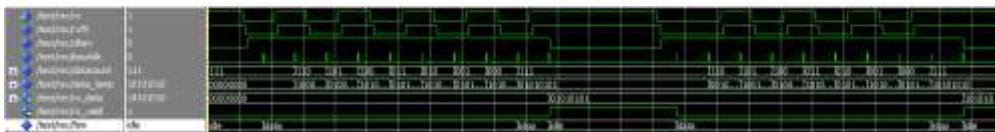
rst      <=      '1', '0' after 100ns;
--rst     <=      '0';

send     <=      '0', '1' after 200ns, '0' after 1us, '1'
after 7 us, '0' after 15us;

end architecture;

```

Jednoduchý testovací příklad, který spojuje vysílač a přijímač. Zvolil jsem mnohem vyšší frekvenci než 9600 Bd, to proto, aby nebylo potřeba tak mnoho simulace.



Pro fyzický test na kitu jsem použil jednoduchou aplikaci, která přijme znak, zneguje nejnižší bit a výsledek pošle zpět. Náš vysílač není „one shot“ – pokud po odvysílání zůstává „tx\_send“ ve stavu 1, vysílájí se stejná data znova. Naopak přijímač drží rx\_valid po celou dobu platnosti dat, až do příchodu dalšího start bitu. Proto nelze spojit rx\_valid a tx\_send napřímo (tedy lze, ale výsledek je trochu jiný, než byste čekali) a je potřeba zapojit tvarovací obvod, který převede vzestupnou hranu na impuls.

## Generátor (pseudo)náhodných čísel

Využijete nejen v hrách, ale i v dalších aplikacích.

## Náhoda

---

Elektronické obvody jsou, pokud jsou navrženy správně, deterministické. Nebo by měly být. To znamená, že pokud je v čase  $T_n$  na vstupech určitá kombinace dat a zároveň známe vnitřní stav obvodu (což zde, na rozdíl od světa elementárních částic, dokážeme), tak můžeme přesně říct, jaký stav bude v čase  $T_{n+1}$ . Díky tomu elektronické obvody fungují tak, jak mají – pokud jsou tedy správně navržené, správně zapojené, správně provozované, správně odstíněné od vnějších vlivů...

Přesto ale někdy potřebujeme něco „znáhodnit“. V té úplně nejjednodušší variantě je to třeba vytvoření elektronické hrací kostky. A tady narazíme na to, že to není úplně taková legrace. Pokud je totiž elektronické zapojení deterministické, lze vždycky zcela přesně říct, jaký bude následující stav, a tedy i jaké číslo „padne“.

Představme si právě tu hrací kostku. Nejjednodušší implementace je pomocí čítače, který čítá hodnoty 1 až 6 (resp. 0 až 5) stále dokola velkou rychlostí, a ve chvíli, kdy hráč zmáčkne tlačítko, tak se čítání zastaví a zobrazí se aktuální stav.

Vidíme, že je tu nějaký generátor sekvence, a pak vnější „znáhodnění“. Kdybychom zůstali jen u toho generátoru a zobrazovali hodnoty v určitých pevně daných intervalech (např. 10 sekund), tak bychom zjistili, že padají stále stejné hodnoty ve stále stejném pořadí. Co s tím?

Jedna možnost je použít opravdový generátor náhodných čísel, což je většinou nějaké hardwarové zařízení, které generuje náhodný signál na základě nějakého fyzikálního jevu. Například rozpad radioaktivní látky, měření šumu nebo vstupu od uživatele. Rozpad radioaktivních látek nebývá pro elektroamatéra naprostě běžně dostupný. S šumem je to lepší. Principiálně vezmeme nějakou součástku, která generuje šum, například polovodičovou diodu, výstupní šum rádně zesílíme, vzorkujeme, převádíme do digitální podoby a podle nejméně významného bitu určujeme aktuální hodnotu náhodného signálu. Nevýhoda je, že šum je ovlivnitelný zvnějšku, například se nám může v obvodu indukovat nějaký radiový signál. Na druhou stranu můžeme takovéto ovlivnění považovat za „chaos zvyšující faktor...“

Pokud to zapojení umožnuje, je dobré použít vstup od uživatele, například stisk tlačítka nebo pohyby myši, a využít toho, že intervaly stisknutí jsou rádově nižší než rychlosť běhu čítače a přicházejí, z hlediska systému,

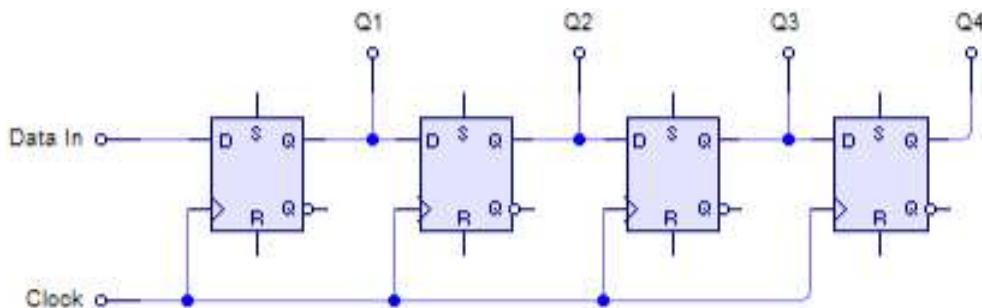
opravdu v náhodné okamžiky. Problém nastane, když potřebujeme náhodná čísla „neustále“, zatímco vstup od uživatele přijde z hlediska systému „za extrémně dlouhou dobu“.

Využívá se techniky, kdy se sice čítají impulzy, ale hodnoty nejdou po sobě v očekávaném pořadí 1, 2, 3, 4, 5, 6..., ale např. u tříbitového čítače jako 1, 4, 6, 7, 3, 5, 2 (hodnota 0 se nevyskytuje). Takovou posloupnost odhalíme hladce, ale představte si, že použijeme čítač šestnáctibitový, který má na výstupu čísla 1 až 65535 v pseudonáhodném pořadí. I tady se posloupnost objeví znova, ale později. Ale nic nám nebrání použít čítače vícebitového. 32 bitů... 60 bitů... 128 bitů... Klidně i 168 bitů. V takovém případě se posloupnost opakuje po cca  $10^{50}$  hodnotách. Pokud použijeme náš 50MHz hodinový signál, objeví se stejná posloupnost po  $7,4 \times 10^{42}$  sekundách, což je  $2,37 \times 10^{35}$  let. Chcete-li to převést na biliony, je to  $2,37 \times 10^{23}$  bilionů let. Vesmír má zatím za sebou 14 bilionů let... To by asi šlo. Teď jen správně nastavit tu hodnotu, od které to spustíme. Ideálně nějakým generátorem náhodných čísel...

Ne, dělám si legraci: do obvodu můžeme zařadit „znáhodňovač“, který v závislosti na nějakém vnějším impulsu změní hodnotu. S následující technikou je to jednodušší, než si myslíte.

### LFSR

Posuvným registrům jsme se ještě nevěnovali nijak důkladně. Přitom jsme je už použili, stačí vzpomenout na serializaci a deserializaci dat. Posuvný registr si představme jako sadu registrů, zapojených za sebou tak, že s každým pulsem hodin se informace z registru N posune do registru N+1. Při převodu paralelních dat na sériová nahrajeme do všech registrů naráz požadovaná data, a pak na výstupu z posledního registru čteme bit po bitu. Při opačném převodu posíláme sériová data na vstup posuvného registru, s každým hodinovým pulsem se posunou o jednu pozici, a jakmile máme načtený kompletní bajt, přečteme si ho z jednotlivých registrů.



Kruhový posuvný registr vznikne tím, že výstup zavedeme zpátky na vstup. Speciálním případem kruhového registru je kruhový posuvný registr s lineární zpětnou vazbou, neboli LFSR (Linear feedback shift register).

Pokud například na předchozím obrázku zavedeme negovaný výstup Q4 na vstup „Data In“, získáme čtyřbitový Johnsonův čítač, který prochází stavy (desítkově): 0, 1, 3, 7, 15, 14, 12, 8.

Cyklus	Q4	Q3	Q2	Q1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0

Cyklus	Q4	Q3	Q2	Q1
6	1	1	0	0
7	1	0	0	0

Kruhové posuvné registry (též „kruhové čítače“) můžeme zapojit i složitěji – vstup budíme nikoli samotným výstupem, ale signálem, složeným z více bitů. Představme si, že u výše uvedeného čítače přivedeme na vstup Data In signál, který vznikne jako Q1 xor Q4.

Cyklus	Q4	Q3	Q2	Q1
0	0	0	0	1
1	0	0	1	1
2	0	1	1	1
3	1	1	1	1

Cyklus	Q4	Q3	Q2	Q1
4	1	1	1	0
5	1	1	0	1
6	1	0	1	0
7	0	1	0	1
8	1	0	1	1
9	0	1	1	0
10	1	1	0	0
11	1	0	0	1

Cyklus	Q4	Q3	Q2	Q1
12	0	0	1	0
13	0	1	0	0
14	1	0	0	0
15	0	0	0	1

Dekadicky zapsané stavy jsou: 1, 3, 7, 15, 14, 13, 10, 5, 11, 6, 12, 9, 2, 4, 8 – vidíme, že takový obvod projde 15 stavy z 16 možných (stav 0000 je konstantní, nijak se nemění). Za cenu určitých úprav zapojení můžeme cyklus rozšířit o poslední stav. Důležité je, že čísla tvoří sice periodu, ale v jejím rámci jsou dostatečně promíchána.

U čtyřbitového čítače můžeme zvolit s dvouvstupovým XOR hradlem šest kombinací zpětné vazby (1,2);(1,3);(1,4);(2,3);(2,4);(3,4). Některé z nich vedou k velmi krátkým cyklům (třeba 1,2), jiné k nejdelším možným (1,4). Sympatické na LFSR je, že k dosažení nejdelšího možného cyklu ( $2^N - 1$  stavů) vystačíme s dvou-, či čtyřvstupovými hradly. I LFSR o délce 168 bitů postavíme jednoduše, zavedením zpětné vazby signálem ( $Q_{151} \text{ xor } Q_{153} \text{ xor } Q_{166} \text{ xor } Q_{168}$  – číslujeme od 1). Vhodné koeficienty najeznete v literatuře – všimněte si, že součástí zpětnovazebního výrazu je vždy nejvyšší bit (pokud by nebyl, degradovali bychom LFSR na menší bitovou šířku).

Pro LFSR existuje poměrně složitý matematický aparát, který dokazuje, že pro libovolnou délku existuje alespoň jedna kombinace vstupů („padů“), která dává nejdelší možný cyklus, a jaká to je, ale její popis je mimo rámec článku. V praxi postačí vědět, že tomu tak

je, a kde najdu vhodné kombinace čísel. Pro délky 20-168 je najdete [zde](#), pro čítače dlouhé 2 až 786 bitů zase [zde](#). BTW, pro čítač o délce 4096 bitů použijte pady 4069, 4081, 4095 a 4096.

Pokud budeme z LFSR odebírat jeden bit (např. nejvyšší), získáme na něm pseudonáhodnou posloupnost 1 a 0, u dlouhých registrů s periodou dostačně dlouhou na to, abychom je prohlásili za náhodné. A do zpětné vazby můžeme přimíchat (*přiXORovat*) právě to vnější „znáhodnění“. Například změnit bit pokaždé, když uživatel zmáčkne tlačítko, nebo když přijde informace po sériové lince – zkrátka cokoli, co se vyskytuje asynchronně a v náhodných intervalech. Změnou bitu posuneme pozici v posloupnosti na jiné místo (jen musíme dávat pozor na kombinaci „samé nuly“, která by běh generátoru zastavila).

Způsob, jaký jsme si popsali, se nazývá „many-to-one“, neboli „mnoho do jednoho“ – několik výstupů se XORuje do jednoho vstupu. Někdy nemusí být toto uspořádání vhodné, např. v případě velmi rychlých obvodů, kde se může projevit zpoždění při víceúrovňovém XORování. V takovém případě můžeme architekturu „many-to-one“ nahradit architekturou „one-to-many“, kde se výstup (nejvyšší bit) přimíchává do několika míst v řetězu klopných obvodů. Místa, kde má dojít ke XORování, jsou stejná jako u architektury many-to-one, sekvence zůstane stejně dlouhá, ale posloupnost hodnot bude jiná.

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

entity lfsr is
port (
    clk: in std_logic;
    q: out std_logic
);
end entity;

architecture many2one of lfsr is
    signal d: std_logic_vector(31 downto 1) := (others=>'1');
begin

    process(clk) is
    begin
```

```

        if rising_edge(clk) then
            for i in d'LOW to d'HIGH-1 loop
                d(i+1) <= d(i);
            end loop;
            d('LOW) <= d(31) xor d(28);
        end if;
    end process;

    q<=d(d'HIGH);

end architecture;

```

Vidíme posuvný registr o šířce 31 bitů (čísluju nikoli od nuly, ale od jedničky). Zápis je dostatečně obecný, takže při úpravě na jinou šířku registru stačí změnit pouze zpětnovazební funkci. Pro zajímavost ještě architektura one-to-many:

```

architecture one2many of lfsr is
    signal d: std_logic_vector(8 downto 1) := (others=>'1');
begin

    process(clk) is

        begin
            if rising_edge(clk) then
                d(1) <= d(8);
                d(2) <= d(1);
                d(3) <= d(2) xor d(8);
                d(4) <= d(3) xor d(8);
                d(5) <= d(4) xor d(8);
                d(6) <= d(5);
                d(7) <= d(6);
                d(8) <= d(7);
            end if;
        end process;

        q<=d(8);

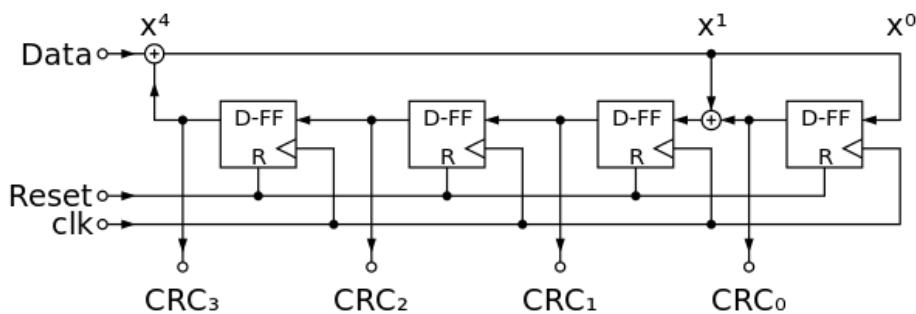
    end architecture;

```

U osmibitového registru je kombinace vstupů s nejdelším cyklem např. (2, 3, 4, 8) – a vidíte, že osmý výstup se XORuje s výstupem na pozicích 2, 3 a 4. Jiná sestava pinů je např. (4, 5, 6, 8).

Pro zajímavost: LFSR (zvané též „polynomial counters“) se používají v obvodech pro generování zvuku ([SID](#), [POKEY](#) apod.) jako šumové generátory.

Pokud do zpětné vazby u architektury one-to-many přimícháme (XOR) serializovaná data, získáme tak nástroj pro výpočet [kontrolního součtu CRC](#) (Cyclic Redundancy Check).



## VGA: Generujeme obraz

Víte, co v mé případě rozhodlo, že se naučím pracovat s FPGA? Byla to právě snadnost generování videosignálu. Tam, kde se u jednočipů a procesorů neobejdete bez specializovaných obvodů nebo velmi přesného časování, najednou nejste s FPGA ničím omezeni! Pojďte se přesvědčit!

Bezpočtu krát jsem po večerech snil o vlastním osmibitu s video výstupem. Asi bych to dokázal nějak udělat pro černobílý PAL signál, ale to není ono. Barevný signál bych asi zvládl taky, i když bych si k němu musel dát nějaké ty AD7xx, co převedou RGB na PAL. Proti tomuto řešení hovořily dva argumenty. Zaprvé: Je s tím spousta práce a pliplačky. Zadruhé: Barevných televizí je čím dál míň, zato VGA monitory se všude válejí už skoro „za odvoz“. Takže VGA. Jenže vytvořit signál pro VGA monitor není až taková brnkačka. U AVR s jeho taktem okolo 20 MHz jste už dost na hraně. Párkrát jsem přemýšlel, jak by to asi bylo složité v případě FPGA, a říkal jsem si, že asi hodně. Ale pak jsem viděl video s tutorialem...

FPGA totiž často obsahují PLL, které umožňují vygenerovat velmi širokou škálu hodinových pulsů. U FPGA máte třeba 48MHz krystal, a díky PLL z toho vygenerujete klidně 51,84MHz (frekvence pro VGA rozlišení 768 x

576). Nebo i 162 MHz ( $1600 \times 1200$ ). Nebo i jiná. Nejste totiž vázáni rychlostí procesoru, ale spíš mezními hodnotami FPGA, a ty jsou dostatečně vysoko. Navíc, jak jsme si už říkali, je FPGA „masivně paralelní“, takže to není tak, že bychom v jednom cyklu museli přečíst data, převést je na RGB, spočítat synchronizaci, ale tohle všechno se děje najednou.

### *Teorie VGA signálu*

---

Videosignál se skládá ze snímků (u prokládaného videa je to komplikovanější o půlsnímky). Každý snímek začíná nějakou synchronizací („Teď jsme na horním okraji obrazovky“), pak je chvíle klidu (zatmění, „back porch“), pak se vykresluje obraz po jednotlivých řádcích, pak je zase zatmění („front porch“), pak synchronizace, back porch, obraz... a tak dále. Tohle celé se děje alespoň 50x za sekundu, u VGA monitorů častěji (60, 72, 75, 85, 100). Je to známá \_obrazová obnovovací frekvence \_a udává se v hertzech.

Snímek se kromě synchronizace a zatmění skládá z obrazových řádků. Možná vám teď bude připadat, že se opakuju, ale: obrazový řádek u VGA začíná synchronizačním pulsem, pak je chvíle klidu (back porch), pak jednotlivé pixely tak jak jsou vedle sebe zleva doprava, pak zase zatmění (front porch), a následuje sync...

Máme tedy osm základních časovacích údajů, čtyři pro horizontální a čtyři pro vertikální. Vždy to je: front porch, sync, back porch a video. U řádků (horizontálně) se udává v jednotkách „pixelů“ (např. „sync puls trvá 120 pixelů“), u snímků (vertikální jednotky) se udává v počtu řádků (např. „vertikální synchronizace trvá 6 řádků“). Někde se udávají tyto počty v mikrosekundách a milisekundách, zejména u videosignálu pro PAL (kde jeden řádek trvá 64 mikrosekundy), ale u VGA je praktičtější udávat je tak, jak jsem napsal, tedy v pixelech a řádcích.

Další důležitý údaj je „pixelová frekvence“. Tedy rychlosť, jakou je potřeba posílat pixely do monitoru. Tato rychlosť je definovaná ve standardech pro jednotlivá rozlišení. Například rozlišení 800 x 600 na 72 Hz používá pixel clock rovno 50 MHz. Tedy na jeden pixel máme  $1/50M = 20\text{ ns}$ . To je naše základní jednotka. Jeden řádek tedy zabere 800 pixelů ( $800 * 20 = 16\text{ }\mu\text{s}$ ) plus neviditelnou část: 56 pixelů front porch, 120 pixelů horizontální synchronizace, 64 pixelů back porch. Tedy 1040 pulsů hlavních hodin =  $20,8\mu\text{s}$ .

Totéž platí i pro vertikální rozlišení. 600 řádků obrazu + 37 řádků front porch + 6 řádků synchronizace + 23 řádků back porch = 666 řádků. Jestliže každý řádek trvá  $20,8\mu s$ , tak vynásobením dostáváme trvání jednoho snímku  $13,8528ms$ , což dává opravdu obnovovací frekvenci  $72 Hz$  (přesně  $72,187572$  periodicky).

### *Synchronizace*

---

Na jednu stranu jsou monitory poměrně tolerantní a odpustí vám drobné (ale jen opravdu drobné) rozladění. Dokážou se i přesto „chytit“. Na jiné signály jsou zase docela citlivé, a vám se tak stane, že koukáte na známé NO SIGNAL. Proto je dobré snažit se dodržet časování co nejlíp to půjde.

Už jsme si řekli, že u řádku i u snímku je vždy oblast aktivních dat a oblast zatmění a synchronizace. Dřív, u CRT monitorů, se zatmění používalo k návratu elektronového paprsku zpátky na levý či horní okraj obrazovky. Každopádně: Po posledním pixelu na řádku stáhněte výstupy R, G, B na nulu. Během front porche se jen čeká. Při horizontální synchronizaci se posílá puls na vodič HSYNC. Na konci se HSYNC vrací do klidu a opět se čeká (back porch). To, jestli je HSYNC normálně 1 a puls 0, nebo jestli je normálně 0 a sync puls 1, je opět určeno standardem (Hsync positive / Hsync negative).

Totéž platí pro vertikální synchronizaci.

Pojďme si teď ukázat entitu sync, která z hodinového cyklu vygeneruje potřebné synchronizační pulsy (hsync, vsync a blank):

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

entity sync is
port (
    clk: in std_logic;
    hsync,vsync: out std_logic;
    blank: out std_logic
);
end sync;

architecture main of sync is
```

```

signal hpos: integer range 0 to 832:=0;
signal vpos: integer range 0 to 509:=0;
begin
    process(clk) is begin
        if rising_edge(clk) then
            if (hpos<832) then hpos:=hpos+1;
            else
                hpos<=0;
                if (vpos<509) then vpos:=vpos+1;
                else vpos<=0;
                end if; --vpos
            end if; --hpos

            if (hpos>32 and hpos<80) then
                hsync<='0';
            else
                hsync<='1';
            end if;

            if (vpos>1 and vpos<4) then
                vsync<='0';
            else
                vsync<='1';
            end if;

            if ((hpos>0 and hpos<192) or (vpos>0 and vpos<29))
then
                blank<='1';
            else;
                blank<='0';
            end if;
        end if; --clk
    end process;
end main; --architecture

```

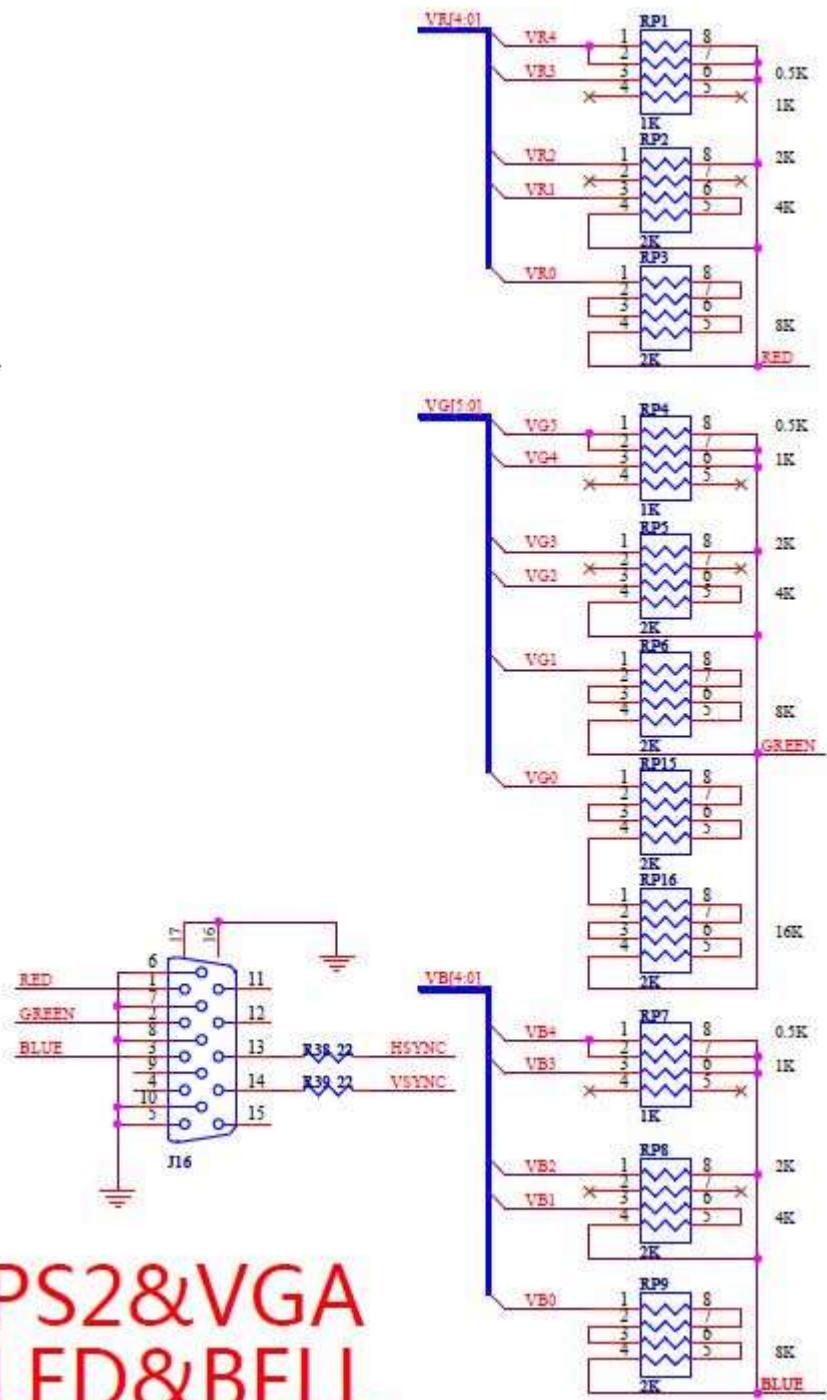
Použil jsem časování pro režim 640×480 na 85 Hz. Horizontální konstanty jsou: 640 pixelů video, 32 front porch, 48 sync, 112 back porch, Hsync negativní. Vertikální konstanty: 480 řádků video, 1 řádek front porch, 3 řádky sync, 25 řádků back porch, Vsync negativní.

V kódu jsou dva čítače, hpos a vpos. Hpos čítá postupně front, sync, back a video, Vpos ve stejném pořadí. Z toho vyplývají i různé „magické konstanty“ v kódu: hpos 32 znamená „konec front porch“, hpos 80 je „front porch + sync“, hpos 192 je celé horizontální zatmění. U vertikálního je to obdobné.

## *R, G, B*

---

Vlastní videosignál je u VGA analogový. Pro takové to domácí použití si vystačíme s prostými několikabitovými odporovými převodníky, pro profesionální zařízení pak použijte specializované rychlé D-A převodníky. Já jsem se chystal spájet takový převodník pro svůj oblíbený kit, ale nakonec jsem [získal velmi levně jiný kit](#), který už má VGA zapojené „z výroby“. Používá šestnáctibitové rozhraní, zapojené jako 5-6-5. Tedy 5 bitů pro červenou, 6 bitů pro zelenou, 5 bitů pro modrou. A jak je vidět ze schématu, moc se s tím nemazali:



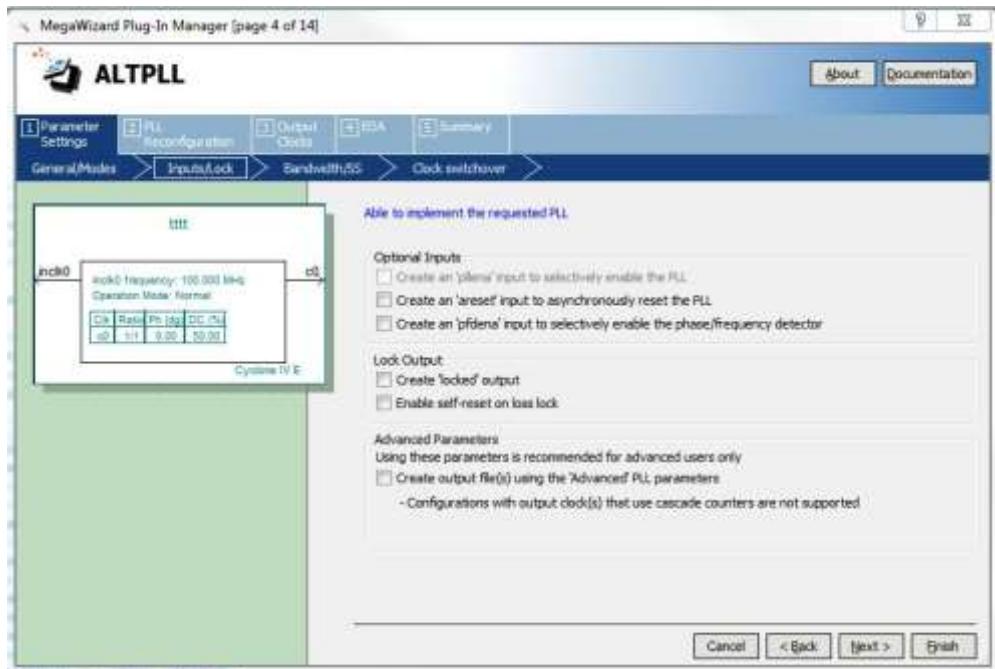
**PS2&VGA  
LED&BELL**

## PLL

Pro generování obrazu ještě potřebujeme ten známý „pixelový kmitočet“. K jeho vygenerování slouží právě PLL. Použití PLL se liší u jednotlivých výrobců FPGA. Já se podržím toho, co nabízí Altera.

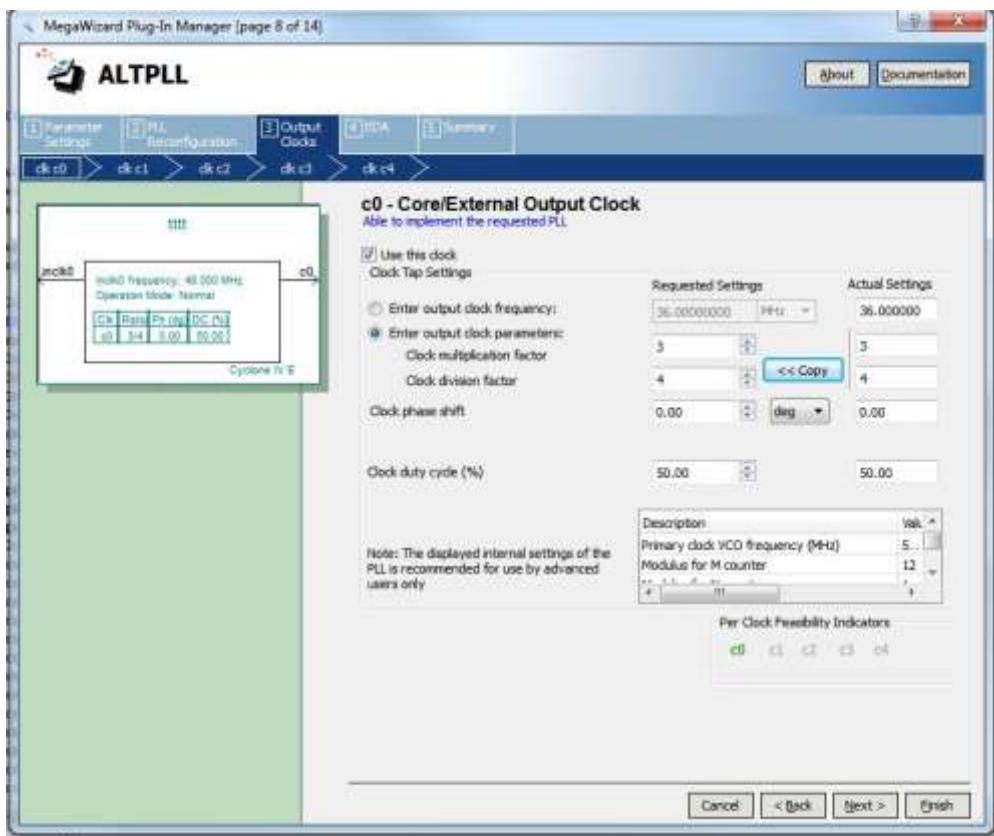
Nejprve vyberte z menu Tools možnost „MegaWizard Plug-In manager“. Nechte „vytvořit novou megafunkci“ a v dalším kroku vyberte „altpll“ (je v oddíle I/O). Jako jazyk vyberte VHDL, vyberte vhodné umístění a jméno souboru („pll“ není špatné) a pokračujte dál v průvodci.

Na straně 3 vyberte rychlosť vašeho FPGA (udává to číslice na konci označení, já mám EP4CE6E22C8N – a rychlosť je právě těch 8). Zadejte vstupní frekvenci v MHz (na mém kitu to je 48), pokračujte dál. V kroku 4 vypněte všechny funkce (reset, locked output apod.), aby bylo PLL co nejjednodušší.



Klikněte klidně na next, next, next, až se dostanete do oddílu 3 – Output Clocks (stránka 8). Zde máte dvě možnosti: Bud' zadáte násobitel a dělitel (celá čísla), nebo požadovanou hodnotu výstupní frekvence (dělitel a násobitel se dopočítají z té zadané vstupní). Pokud budu implementovat rozlišení 640×480@85Hz, budu potřebovat frekvenci 36 MHz, kterou získám

ze vstupních 48 prostým vynásobením zlomkem  $\frac{3}{4}$ . Získám i další frekvence, např.  $25/24$  mi dá 50 MHz,  $27/8$  zase 162 MHz ( $1600 \times 1200 @ 60\text{Hz}$ ).



Další hodinové výstupy nebudeme potřebovat, nepotřebujeme ani specifický fázový posun nebo duty cycle, takže můžeme klidně použít Finish. Wizard vygeneruje několik souborů a vloží je do projektu (popř. se zeptá, jestli to má udělat).

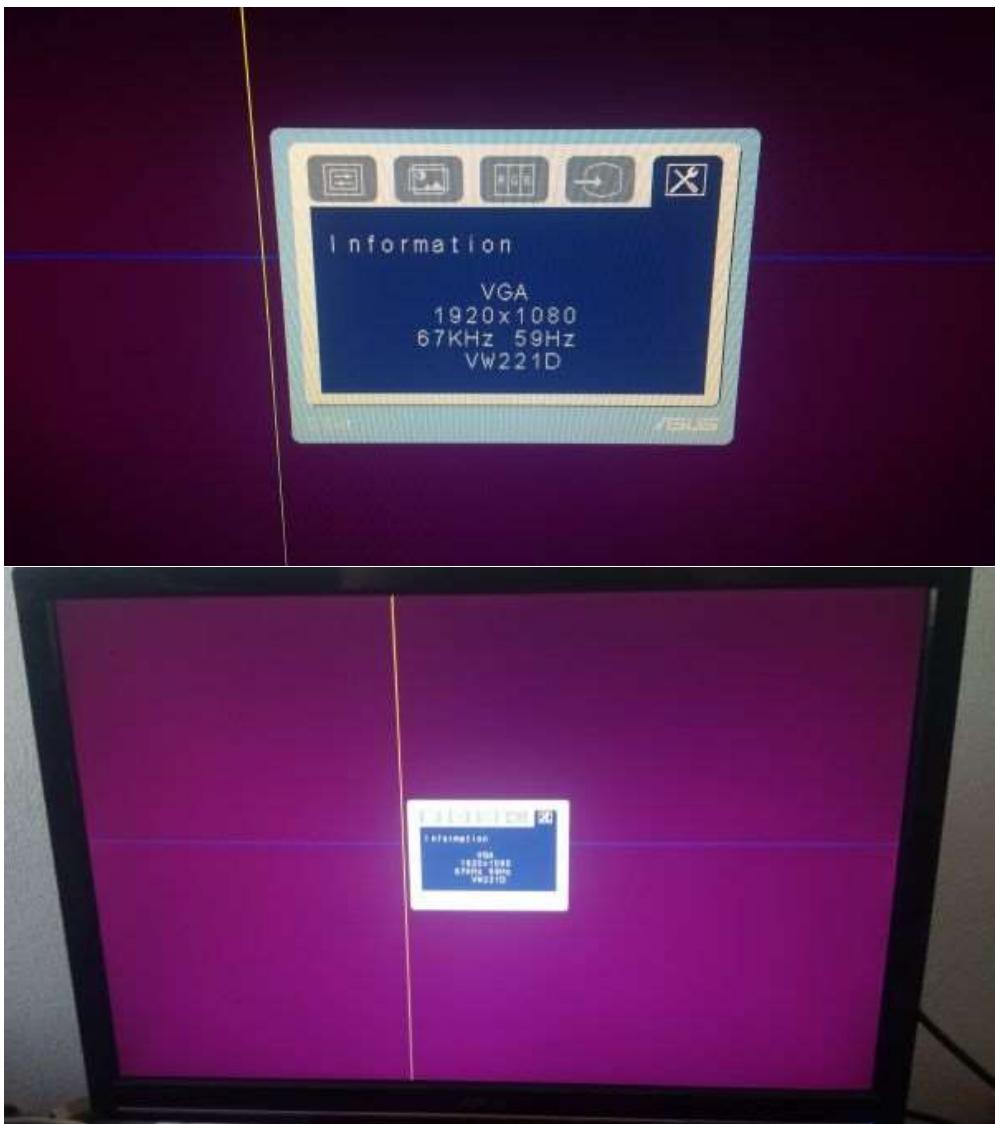
### Kalkulačka!

Jestli se vám motá hlava z nadmíry konstant a frekvencí, tak nezoufejte! Připravil jsem pro vás [kalkulátor frekvencí](#) pro VGA, PLL a vůbec všechno to, co jsme si teď říkali. Stačí zadat frekvenci krystalu, kterou máte k dispozici, a kliknout na Go!

V tabulce se objeví přehledně všechna VGA rozlišení, která lze z této hodinové frekvence získat. K nim všechny potřebné konstanty, a dokonce i nastavení PLL (násobitel a dělitel). Kalkulačka vždy počítá frekvence tak, aby násobitel a dělitel byla celá čísla. Jejich velikost si můžete omezit v poli *PLL max div* (např. pro jednodušší PLL).

Když si vyberete vhodné rozlišení a kliknete na něj, tak vám [kalkulačka](#) vytvoří generuje vylepšenou komponentu sync.vhd (viz výše) a ukáže aktuální hodnoty průběhu signálů.

Při svých testech jsem se odvážně pouštěl dál a dál, a nakonec jsem použil rozlišení 1920×1080@60Hz, tedy Full HD. Pixelová frekvence je 148,5MHz, PLL koeficient 99/32 (a opravdu,  $48 * 99 / 32 = 148,5$ ). S trochu rozechvění jsem nahrával kód do FPGA, a po několika sekundách se na monitoru objevilo:



Jak se říká: *Bylo to tam!*

### *Jednoduchý obrazec*

---

Synchronizace je nezbytný základ, je to kostra, na který se navěší vlastní zobrazování. Vylepšená komponenta sync posílá kromě synchronizačních

pulsů a signálu blank i dvě čísla, totiž posx a posy, neboli pozici horizontálně a pozici vertikálně. Funkce je následující: Když je zatmění (blank), tyto čísla ignorujte a na výstupy R,G,B posílejte 0. Pokud není zatmění, posílejte na výstupy barvu pixelu na dané souřadnici.

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

ENTITY VGA IS

port (
  clock48: in std_logic;
  vga_hs,vga_vs: out std_logic;
  vga_r: out std_logic_vector(4 downto 0);
  vga_g : out std_logic_vector(5 downto 0);
  vga_b: out std_logic_vector(4 downto 0)
);

END VGA;

architecture main of vga is

signal vgclk:std_logic:='0';
signal x:integer range 0 to 1919;
signal y:integer range 0 to 1439;
signal blank: std_logic;

-----
component PLL1 is
  port (
    inclk0      : IN STD_LOGIC  := '0';
    c0          : OUT STD_LOGIC
  );
end component PLL1;
-----
component sync is
port (
  clk: in std_logic;
  posx:out integer range 0 to 1919;
  posy:out integer range 0 to 1439;
  hsync,vsync: out std_logic;
  blank: out std_logic
);
end component sync;
```

```

begin
c1: sync port map (vgaclk, x, y, vga_hs, vga_vs, blank);
c2: pll1 port map (clock48, vgaclk);

process (vgaclk) begin
    if rising_edge(vgaclk) then

        -- obrazec
        if (x > 950 and x<970) then
            -- svisle barevne prouzky uprostred
            vga_r<=std_logic_vector(to_unsigned(y mod 32,5));
            vga_g<="111111";
            vga_b<="00000";
        elsif (y > 530 and y<550) then
            -- horizontalni cara
            vga_r<="00000";
            vga_g<="000000";
            vga_b<="11111";
        elsif (blank='0') then
            -- kde neni nic, tam je pozadi
            vga_r<="00111";
            vga_g<="000000";
            vga_b<="00111";
        elsif (blank='1') then
            -- zatmeni
            vga_r<=(others=>'0');
            vga_g<=(others=>'0');
            vga_b<=(others=>'0');
            end if;

        end if; --clk
    end process;

end main;

```

Komponentu PLL1 mi vygeneroval výše zmíněný MegaWizard. Komponentu sync jsem si nechal vygenerovat svůj [kalkulačkou](#). A zbytek už je jen „když je pozice taková a onaká, tak nastav barvu na ...“ Vlastně taková obdoba „hello world“ pro VGA.

Šlo by použít místo VGA třeba HDMI? Šlo. Bylo by samo sebou zapotřebí upravit výstup, protože HDMI používá digitální sériový přenos údajů a diferenciální budiče, ale základní princip zůstane stejný, a díky paralelní podstatě fungování FPGA nás převod hodnot na HDMI nijak „nezpomalí“.



# OMEN Alpha, tentokrát ve FPGA

## ToDo



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

**Podpořte její vznik?**

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na  
<https://www.osmibity.cz/addons.html>

# OMEN Echo

# OMEN Foxtrot

ToDo



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

**Podpořte její vznik?**

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na  
<https://www.osmibity.cz/addons.html>

# Vlastní mikroprocesor

## Znáte hru MHRD?

Nebo jinak – znáte „programátorské“ a „inženýrské“ hry od Zachtronics? Například TIS-1000, Infinifactory nebo Shenzhen I/O? V těchto hrách neběháte, nestřílíte a nedobýváte území, ale konstruujete elektronické obvody a tvoříte automaty, které dělají určité procesy.

Hra MHRD (Micro Hard) před hráče staví několik příkladů, stylem „puzzle“. Dostanete zadání a v jazyce, podobném VHDL, vytvoříte elektronický obvod. Když jste s ním spokojeni, odešlete ho k otestování, a pokud vyhovuje, pokračujete dál.

A nakonec složíte, vězte nebo ne, funkční mikroprocesor.

Ne že by neexistovaly doslova desítky podobných procesorů. Existují a nazývají se „softcore microprocessors“. Některé z nich implementují reálné procesory (Z80, 6502, AVR apod.), jiné jsou naprostě proprietární a v podobě reálného čipu nikdy neexistovaly.

Velmi vydatný seznam najdete na Wikipedii nebo na známém webu Opencores.org.

[https://en.wikipedia.org/wiki/Soft\\_microprocessor](https://en.wikipedia.org/wiki/Soft_microprocessor)

Některé z těchto soft procesorů jsou velmi komplexní. Některé jsou optimalizované na rychlosť, jiné na masivní paralelní práci, některé jsou postavené na reálném procesoru...

Procesor MHRD – říkejme mu tak – je velmi jednoduchý šestnáctibitový mikroprocesor. Nepoužívá instrukce v tom smyslu, v jakém jsme je poznali u ostatních mikroprocesorů. V podstatě má pouze dvě instrukce. Jedna z nich načítá konstantu do vnitřního registru a druhá provádí aritmetické a logické operace. U druhé můžete pomocí jednoho bitu navíc nastavit, jestli se po provedení instrukce má provést v případě nulového výsledku skok.

Podobná koncepce není nic nového a překvapivého. V dobách před mikroprocesory se používaly takzvané *mikroprogramované řadiče*. V podstatě velmi jednoduché stavové automaty, které četly instrukce z paměti a prováděly jednoduché operace. Instrukce mívaly velkou šířku (klidně 18 bitů i více) a každá skupina bitů řídila nějakou část obvodu. Samotný tok programu („program flow“) řídil jeden nebo několik bitů instrukčního slova. Pokud tyto bity byly nastaveny, tak řadič po vykonání instrukce nezvedl počítadlo o 1, ale přiřadil novou hodnotu.

Pokud vám to připomíná moderní koncepty, jako jsou RISC nebo VLIW, jste na správné stopě. Podobné mikroinstrukce jsou totiž velmi rychlé, jednoduché a mocné. Mnohé procesory jsou dokonce konstruované tak, že se navenek tváří jako CISC, třeba 6809 nebo Pentium, ale uvnitř se jednotlivé instrukce překládají na kratičké mikroprogramy.

## Architektura mikroprocesoru

Nechci tím obtěžovat nijak víc, než je nutné, tak jen připomenu, že mikroprocesor integruje dvě základní části číslicového počítače, totiž aritmeticko-logickou jednotku pro výpočty a operace s daty, a řadič, který řídí a organizuje tok programu a podle načteného operačního kódu provádí nezbytné kroky.

Aritmeticko-logická jednotka (ALU) není žádné mysteriozní zapojení, je to prostá kombinační logika. Většinou má dva vícebitové vstupy, A a B, a řídicí vstupy, které určují, jaká operace se má provést. Většinou nechybí základní operace: sčítání, odčítání, posuvy a rotace, and, or, xor. S ALU souvisí i příznaky – Zero, Carry, Sign, ...

Mimochodem, víte, jak ze sčítačky udělat odčítačku? Pokud potřebujete provést operaci  $A - B$ , bude výsledek stejný, jako byste provedli  $A + /B + 1$  ( $/B$  je původní hodnota B se všemi bity invertovanými).

Řadič je poněkud složitější, tam už si s kombinační logikou nevystačíme. Součástí řadiče je sekvencér, který má na starosti udržování informace o adrese paměti, z níž se čtou instrukce a konstanty, tedy známý „program counter“ PC. Jenže načtení instrukce z nějaké adresy je jen první krok. Nyní je potřeba instrukci dekódovat a provést patřičné operace. Každá instrukce procesoru představuje sekvenci několika kroků. Například v kroku 2 je potřeba nastavit vstupy sčítačky tak, aby na vstup A šel akumulátor a na vstup B obsah nějakého registru, v kroku 3 se výsledek zachytí do pracovního registru a v kroku 4 se z tohoto registru zapíše zpět do akumulátoru.

K rozhodnutí „co se k čemu připojuje“ slouží vícevstupové (de)multiplexory. Pomocí nich přestaví procesor „cestu informací“ podobně jako výhybky na železniční trati nastaví volnou cestu ze třetího nástupiště k severnímu výjezdu...

Práce řadiče a instrukčního dekodéru tedy spočívá především v tom, aby pro každý krok každé instrukce byly správně nastavené „výhybky na trati“. Používalu se v zásadě dvě cesty, jak to zařídit: obvodový řadič a mikroprogramový řadič.

Obvodový řadič si můžeme představit jako čítač jednotlivých kroků, za kterým je připojený obří dekodér, který pro každou možnou kombinaci „instrukční kód x krok“ vygeneruje správně nastavení multiplexorů.

V příloze popisuji princip fungování řadiče procesoru 6502. Tam byla v roli dekodéru použita paměť s organizací 130 řádků po 21 bitech. To byly možné operace. Každý řádek v sobě nesl informaci o tom, za jakých podmínek se má daná operace vykonat. Tedy v kterém kroku, a jak má vypadat instrukční kód. Pokud kombinace vyhovovala, řádek se aktivoval a kombinační logika zařídila podstatné.

Čítač prostě jen čítal takty, a poslední kombinace dané instrukce se postarala o to, že čítač jel opět od nuly.

Mikroprogramové řadiče používají rovněž čítač, ale tentokrát má i schopnost nastavení nějaké hodnoty a funguje jako ukazatel do paměti mikroprogramu. Mikroprogramové instrukce jsou velmi jednoduché, je jich třeba jen několik základních, většinou jen „nastav multiplexory tak a tak“ a „pokud je splněna podmínka, tak skoč na tuto adresu“. Nezapomínejme, že mikroprogram nemá za úkol dělat žádné složité operace, jen „nastavit cestu datům“.

Mikroprogramové řadiče přinášejí proti obvodovým podstatné zjednodušení konstrukce. Teoreticky mohou být v některých ohledech pomalejší, na druhou stranu tuto nevýhodu více než vyrovnávají snazší opravou návrhových chyb. Tam, kde stačí změnit několik mikroinstrukcí, tam je potřeba u obvodového řadiče složitě měnit celou kombinační logiku.

Při návrhu vlastního mikroprocesoru ve VHDL můžete využít oba přístupy. Dostupné implementace reálných mikroprocesorů používají oba přístupy, takže se můžete setkat třeba s implementací procesoru 8080, řízenou mikroprogramem (jmenuje se „light8080“). Doporučuju se podívat na jeho zdrojový kód, je velmi dobře dokumentovaný, včetně zdrojového kódu mikroinstrukcí.

## Přípravné práce

V MHRD, jako v každé správné hře, začínáte od jednoduchých úkolů, od přípravy hradel a kombinačních obvodů.

Nechci spoilovat, proto varuji: Pokud jste MHRD nehráli a chystáte se na to, další obsah kapitoly přeskočte.

Začínáte obyčejným hradlem NAND a jeho variantou, NAND4B. Tato varianta vlastně funguje jako obvod 7400: Čtyři hradla NAND, 2x4 bity jako vstup, 4 bity jako výstup. Následuje invertor NOT, jeho čtyřbitová varianta NOT4B a šestnáctibitová varianta NOT16B.

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

ENTITY nand4b IS
port (
    in1: in std_logic_vector (3 downto 0);
    in2: in std_logic_vector (3 downto 0);
    y: out std_logic_vector (3 downto 0)
);
END;

architecture main of nand4b is begin
y(0) <= in1(0) NAND in2(0);
y(1) <= in1(1) NAND in2(1);
y(2) <= in1(2) NAND in2(2);
y(3) <= in1(3) NAND in2(3);
end architecture;
```

Ekvivalentně předchozím krokům navrhnete hradla AND, AND4B, AND16B, NOR, NOR4B a NOR16B, OR, OR4B a OR16B. Tip: Použijte vektorové operace:

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

ENTITY nor4b IS
port (
    in1: in std_logic_vector (3 downto 0);
    in2: in std_logic_vector (3 downto 0);
    y: out std_logic_vector (3 downto 0)
);
```

```

END;

architecture main of nor4b is begin
y <= in1 NOR in2;
end architecture;

ENTITY nor16b IS
port (
  in1: in std_logic_vector (15 downto 0);
  in2: in std_logic_vector (15 downto 0);
  y: out std_logic_vector (15 downto 0)
);
END;

architecture main of nor16b is begin
y <= in1 NOR in2;
end architecture;

```

Kromě násobných elementů v jednom (OR4B jsou vlastně čtyři OR hradla v jedné komponentě) jsou zapotřebí i vícevstupová hradla, např. OR4W (4-way). Hradlo OR4W má 4 vstupy a jeden výstup. Výstup je v logické 1, pokud je alespoň jeden vstup v log. 1. Analogicky je vytvořeno hradlo OR16W.

Série pokračuje hradlem XOR a jeho variantami XOR4B a XOR16B.

## *Sčítáčka*

---

Po sestavení těchto základních kamenů přichází čas na oblíbené kousky: HALFADDER, FULLADDER a rozšíření: ADDER4B a ADDER16B. Už bych se opakoval, ale pro jistotu:

```

library ieee;
use ieee.std_logic_1164.all;

entity adder is
port(
  i0, i1 : in std_logic;
  ci : in std_logic;
  y: out std_logic;

```

```

        co : out std_logic
    );
end adder;

architecture main of adder is
begin
    y <= i0 xor i1 xor ci;
    co <= (i0 and i1) or (i0 and ci) or (i1 and ci);
end main;

-----
library ieee;
use ieee.std_logic_1164.all;

entity adder16b is
port (
    in1, in2 : in std_logic_vector(15 downto 0);
    carryIn : in std_logic;
    y : out std_logic_vector(15 downto 0);
    carryOut : out std_logic
);
end adder16b;

architecture main of adder16b is

component adder
    port(
        i0, i1 : in std_logic;
        ci : in std_logic;
        y: out std_logic;
        co : out std_logic);
    end component;

signal carry : std_logic_vector(15 downto 0);

begin
    adder_0: adder port map (in1(0), in2(0), carryIn,
y(0), carry(0));
    adder_1: adder port map (in1(1), in2(1), carry(0),
y(1), carry(1));
    adder_2: adder port map (in1(2), in2(2), carry(1),
y(2), carry(2));
    adder_3: adder port map (in1(3), in2(3), carry(2),
y(3), carry(3));
    adder_4: adder port map (in1(4), in2(4), carry(3),
y(4), carry(4));

```

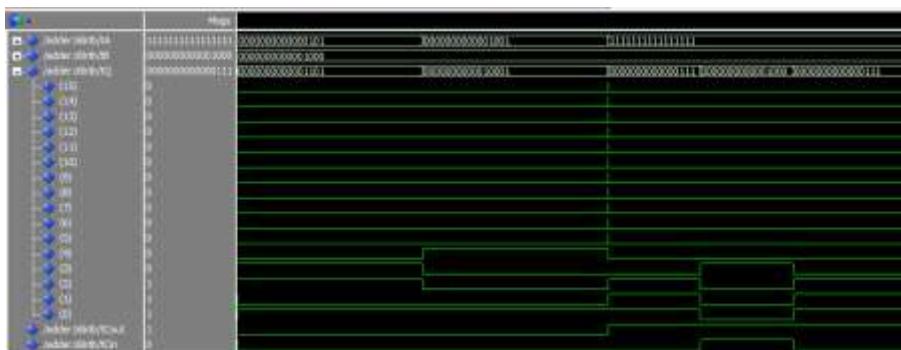
```

        adder_5: adder port map (in1(5), in2(5), carry(4),
y(5), carry(5));
        adder_6: adder port map (in1(6), in2(6), carry(5),
y(6), carry(6));
        adder_7: adder port map (in1(7), in2(7), carry(6),
y(7), carry(7));
        adder_8: adder port map (in1(8), in2(8), carry(7),
y(8), carry(8));
        adder_9: adder port map (in1(9), in2(9), carry(8),
y(9), carry(9));
        adder_10: adder port map (in1(10), in2(10),
carry(9), y(10), carry(10));
        adder_11: adder port map (in1(11), in2(11),
carry(10), y(11), carry(11));
        adder_12: adder port map (in1(12), in2(12),
carry(11), y(12), carry(12));
        adder_13: adder port map (in1(13), in2(13),
carry(12), y(13), carry(13));
        adder_14: adder port map (in1(14), in2(14),
carry(13), y(14), carry(14));
        adder_15: adder port map (in1(15), in2(15),
carry(14), y(15), carryOut);

end main;

```

## Předvídání přenosu



## Multiplexor

Po sčítačce přichází na řadu multiplexor MUX – dva jednobitové vstupy, jednobitový výstup a jeden řídicí bit, který přepíná mezi vstupy.

Roztažením na 4 byty získáme komponentu MUX4B, dalším rozšířením pak MUX16B.

V MHRD je nejvhodnější složit čtyři MUX4B na jeden MUX16B, ale ve VHDL samozřejmě použijeme variantu, v níž se popisuje chování:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux16b is
    port ( sel : in STD_LOGIC;
           in1 : in STD_LOGIC_VECTOR (15 downto 0);
           in2 : in STD_LOGIC_VECTOR (15 downto 0);
           y: out STD_LOGIC_VECTOR (15 downto 0)
        );
end mux16b;

architecture main of mux16b is
begin
    y <= in1 when (sel = '1') else in2;
end main;
```

Pokud přidáme další řídicí bit, získáme čtyřcestný šestnáctibitový multiplexor MUX4W16B.

Analogicky zkonztruujeme demultiplexor DEMUX a jeho čtyřcestnou variantu DEMUX4W.

## ALU

---

Aritmeticko-logická jednotka ALU4B pracuje se čtyřbitovými operandy a čtyřbitovým řídicím slovem. Bity řídicího slova mají tento význam:

- BIT 0: Pokud je 1, je výstup negován.
- BIT 1: Pokud je 0, je na výstupu součet A a B, pokud je 1, je na výstupu výsledek funkce A AND B.
- BIT 2: Pokud je 1, je vstup B negován.
- BIT 3: Pokud je 1, je vstup A negován.

Kromě čtyřbitového výsledku dává ALU i dvě informace navíc – příznak ZERO a příznak NEGATIVE. ZERO říká, že výsledek je nulový, NEGATIVE je hodnota nejvyššího bitu (vlastně znaménko výsledku).

Rozšířením vznikne ALU16B – funkce je stejná, jen operandy a výsledek mají 16 bitů.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity alu16b is
port (in1, in2 : in std_logic_vector(15 downto 0);
      opcode : in std_logic_vector(3 downto 0);
      y : out std_logic_vector(15 downto 0);
      zero, negative : out std_logic);
end alu16b;

architecture main of alu16b is

component adder16b
    port (
        in1, in2 : in std_logic_vector(15
downto 0);
        carryIn : in std_logic;
        y : out std_logic_vector(15 downto 0);
        carryOut : out std_logic
    );
end component;

component mux16b is
    port (
        sel : in STD_LOGIC;
        in1 : in STD_LOGIC_VECTOR (15 downto 0);
        in2 : in STD_LOGIC_VECTOR (15 downto 0);
        y: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;

for adder16b_0 : adder16b use entity
work.adder16b;
for mux16b_0 : mux16b use entity work.mux16b;

signal in1_work : std_logic_vector(15 downto 0);
signal in2_work : std_logic_vector(15 downto 0);
signal out_work : std_logic_vector(15 downto 0);
signal nandout : std_logic_vector(15 downto 0);
signal addout : std_logic_vector(15 downto 0);
signal out_mux : std_logic_vector(15 downto 0);
```

```

begin

    in1_work <= (in1 xor x"0000") when opcode(3) = '0'
else (in1 xor x"FFFF");
    in2_work <= (in2 xor x"0000") when opcode(2) = '0'
else (in2 xor x"FFFF");

    -- Výpočet obou mezivýsledků
    nandout <= in1_work nand in2_work;
    adder16b_0: adder16b port map (in1_work, in2_work,
'0', addout, open);

    -- Výběr mezivýsledku
    mux16b_0 : mux16b port map (opcode(1), nandout,
addout, out_mux);

    -- Negace výstupu?
    out_work <= (out_mux xor x"0000") when opcode(0) =
'0' else (out_mux xor x"FFFF");

    -- Výsledkové příznaky
    negative <= out_work(15);
    zero <= '1' when out_work = x"0000" else '0';

    y <= out_work;
end main;

```

## *Registr*

---

Následují sekvenční obvody. V MHRD se pracuje s tím, že hodinové pulsy jsou „všudypřítomné“ a „globální“, takže je není potřeba explicitně změňovat. Proto má třeba klopný obvod D (DFF) jednobitový vstup a jednobitový výstup; hodiny jsou implicitní.

Komponenta REG (registr) je podobná obvodu DFF, ale navíc má vstup LOAD. Pokud je LOAD=1, zapíše se do obvodu hodnota na vstupu, jinak obvod drží předchozí hodnotu. A opět si připravíme varianty REG4B a REG16B.

Registry si složíme do šestnáctibitové paměti se čtyřmi buňkami RAM4W16B a do obří paměti 64k x 16: RAM64KW16B.

## Čítač

Užitečná komponenta je čtyřbitový čítač COUNTER4B. Funguje tak, jak jsme zvyklí, tedy při každém pulsu hodin zvýší svůj stav o 1. Navíc má vstup RESET, který jej nuluje, čtyřbitový vstup dat a vstupní signál LOAD, který slouží k nastavení vnitřní hodnoty. Rozšířením pak získáme COUNTER16B, který může sloužit jako programový čítač PC.

## Mikroprocesor MHRD

Mikroprocesor ve hře MHRD je šestnáctibitový s velmi jednoduchou instrukční sadou. Obsahuje tři registry:

- Registr PC, což je známý programový čítač
- Registr M, který slouží k výběru adresy v paměti
- Registr A, který slouží jako akumulátor



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

Podpoříte její vznik?

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na <https://www.osmibity.cz/addons.html>

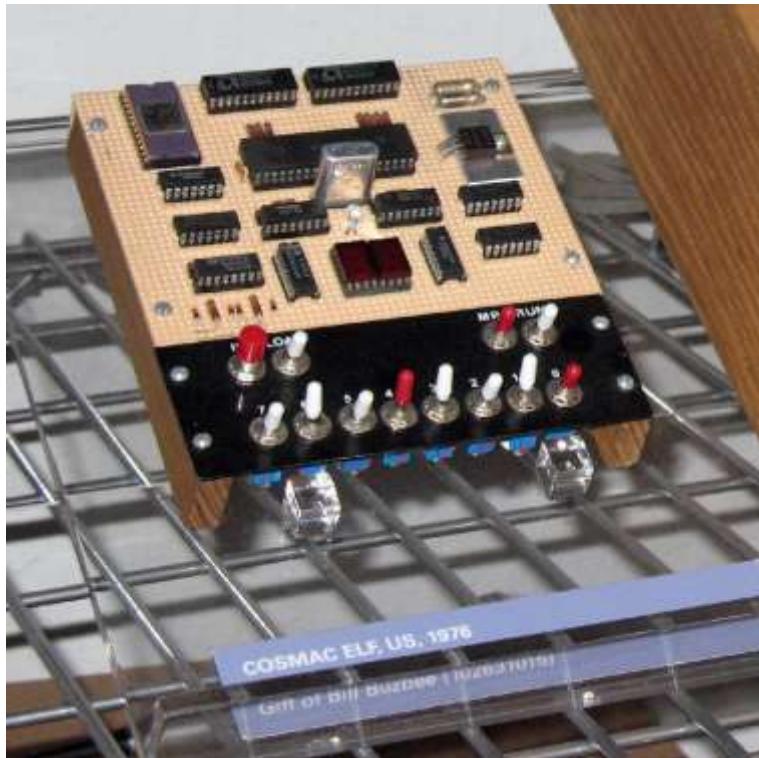
[https://github.com/riktw/Microhard\\_CPU](https://github.com/riktw/Microhard_CPU)

# Z historie mikroprocesorů



Nedá mi to, abych sem tuhle kapitolu nezařadil. Podívejme se na některé kuriózní procesory a na to, jak se tehdy vlastně osmibitové procesory vyráběly.

## Kuriozity



## RCA CDP1802

V roce 1976 uvedla RCA na trh mikroprocesor CDP1802. Byl vyrobený technologií CMOS, vyvinutou právě v RCA pod názvem COS-MOS (COmplementary Silicon / Metal-Oxid-Semiconductor). 1802 byl známý taky pod názvem COSMAC. Prý „Complementary Symmetry Monolithic Array Computer“, ale spíš bych si tipl, že se tam odráží ta technologie COS-MOS.

V letech 1970-1971 vyvinul Joseph Weisbecker vlastní architekturu mikroprocesoru. RCA vytvořila nejprve (v roce 1975) dva čipy 1801R a 1801U, a později je sloučila do jednoho obvodu 1802.

CDP1802 má několik vlastností, které jej výrazně odlišují od hlavních konkurenčních mikroprocesorů té doby (6800, 6502, 8080, Z80). Například jádro CDP1802 je statické – znamená to tedy, že je možné zpomalit hodiny na minimum, dokonce je i zastavit, a procesor neztratí data z vnitřních registrů.

CDP1802 adresuje, podobně jako ostatní mikroprocesory té doby, 64 kB paměti. Uvnitř obsahuje šestnáct šestnáctibitových registrů (R0-R15), které jsou, až na dvě výjimky, plně ortogonální. Akumulátor, v němž se provádějí aritmetické operace, je osmibitový a je označený D. Procesor obsahuje i dva čtyřbitové registry P a X. Registr X udává, který z registrů R0-R15 funguje jako indexový registr pro přístup k datům, registr P pak říká, který z registrů R0-R15 funguje jako programový čítač. Ano, CDP1802 nemá dedikovaný registr PC (Program Counter), místo něj používá vždy jeden z registrů R0-R15.

Je to celé trošku divoké a zvyknout si na to dá člověku, zvyklému na architekturu 8080/6800 a odvozené, trochu práce. Tak například procesor nemá zásobník. Na co taky, když nemá instrukci CALL. Zato má hned dvě instrukce RET – jedna při návratu povolí přerušení, druhá ho zakáže. Jak se volá podprogram? No, do nějakého registru si dáte jeho adresu, a pak řeknete: Ty teď budeš program counter! A na konci řeknete: Program counter je zase ten původní...

Samozřejmě, existují instrukce, které uloží „něco jako stav“ – tedy registry P a X – na „něco jako zásobník“, tedy do paměti, adresované buď registrem RX (tedy registrem, jehož číslo je v registru X), nebo registrem R2. Tohle je jedna výjimka z ortogonalnosti. Další jsou, že registr R0 je používán pro přenos DMA, registr R1 je program counter při přerušení a jediná instrukce, kterou by šlo považovat za jakous-takous instrukci pracující se zásobníkem, používá k adresování R2.

Procesor CDP1802 má i jeden jednobitový datový výstup (Q) a čtyři jednobitové datové vstupy (EF1-EF4). Instrukční sada obsahuje instrukce Set Q (SEQ) a Reset Q (REQ). Pro testování vstupů pak jsou podmíněny skoky B1, B2, B3, B4, BN1, BN2, BN3, BN4 – skáče se, pokud je vstup = 1, resp. pokud je roven 0 (u varianty BN).

Skoky jsou osmibitové, ale nikoli relativní. Pouze se vezme osmibitová adresa a vloží se do nižšího byte registru PC (= registru R, jehož číslo je v registru P... Zvykejte si!) Hranici 256 byte normálním (krátkým) skokem

nepřeskocíte. Naštěstí existují i instrukce dlouhého skoku s absolutní adresou.

Ke skokům připočtěme i instrukce „přeskoků“, včetně podmíněných – tož instrukce, které umožňují přeskocit následující jeden nebo dva byty.

Když píšu „podmíněné“, tak si nepředstavujte nějakou podmínkovou bonanzu. Kdepak. Podmínky jsou buď podle datových vstupů EF, podle stavu výstupu Q, nebo podle toho, jestli je registr D roven nule. Jediný podmínkový příznak je DF, a ten odpovídá příznaku přenosu (CY). Testovat můžete stav příznaku povoleného přerušení (IE), a to instrukcí LSIE. Pokud je IE=1, přeskocí se následující dva bajty.

Existují instrukce pro přesun dat mezi pamětí a registrem D, adresované buď konkrétním registrem, nebo registrem, jehož číslo je v X. Jsou i varianty s postinkrementací a postdekrementací. Na druhou stranu nejsou instrukce pro přímou práci s registry R – pokud chcete do registru uložit nějakou hodnotu, musíte ji ukládat nadvakrát přes registr D. Instrukce PLO, PHI (Put Low / High) a GLO, GHI (Get Low / High) přenesou data mezi registrem D a dolní, resp. horní polovinou vybraného registru.

Procesor má navíc možnost přímo adresovat sedm vstupní / výstupních portů (instrukcemi INP 1 – INP 7 a OUT 1 – OUT 7), kdy probíhá přenos z / do paměti na adresu, která je v RX. Kromě toho oplývá i zabudovaným řadičem DMA: po spuštění přenáší byty z / do paměti, adresované registrém R0.

Říkáte si „Kdo by něco takového, proboha, použil?“ Inu, tehdy ještě nebyla v lidech zafixovaná architektura 8080/6800, tak měli možná otevřenější přístup. Snad. Doufám. Po výše uvedeném popisu vás jistě nepřekvapí, že první vyšší programovací jazyk pro tento procesor byl FORTH. Na druhou stranu má CDP1802 některé výhody, především pak možnost zastavit hodiny procesoru a snížit tak jeho odběr na naprosté minimum. Jeho instrukční sada má, i přes všechny podivnosti, několik zajímavých rysů: Naprostá většina instrukcí trvá dva strojové cykly (každý 8 taktů hodin), až na vzácné výjimky, které trvají 3 cykly. Počítání času je tedy velmi jednoduché.

Minimální odběr oceníte, když děláte třeba vesmírnou sondu. V mnoha sondách právě tento procesor je. Ostatně, od Intersilu si jej můžete objednat ve variantě se zvýšenou odolností proti radiaci. Jen se připravte na to, že to není nejlevnější špás...

CDP1802 byl použit v legendárních počítačích COSMAC ELF a COSMAC VIP, v herní konzoli RCA System II, a využívaly ho i jugoslávské počítače Pecom 32 a Pecom 64.

RCA vyráběla vylepšenou variantu CDP1804 – tento procesor uměl více instrukcí (třeba i CALL), obsahoval i časovač a měl integrovanou RAM a ROM. Verze CDP1805 byla bez ROM, verze CDP1806 i bez RAM.

## Klony



První mikroprocesor, a to je už notoricky známé, byl Intel 4004. Intel ho vyráběl ve třech verzích – tedy navenek. Uvnitř to byl stejný čip, lišilo se jen pouzdro – plastické, keramické, nebo plasto-keramické. Kromě Intelu ale vyráběl stejný procesor i National Semiconductor (NatSemi) pod označením INS4004. Stal se tak *druhým zdrojem* (second source).

„Druhý zdroj“ byl u integrovaných obvodů poměrně běžný koncept. Například známou TTL řadu 74xx představil výrobce Texas Instruments (obvody nesly označení SN74xx). Jenže kromě TI začaly vyrábět funkčně ekvivalentní obvody i další výrobci – namátkou Fairchild, zmíněný NatSemi, Philips, za železnou oponou to byla například Tesla (MH74xx), polská Unitra (UCY74xx), vyráběly se i v Maďarsku (Tungsram), Rumunsku i ve východním Německu (VEB Kombinat Mikroelektronik Erfurt – KME, např. německý D174 byl ekvivalent SN7474). Většinou se výrobci snažili, aby stejně označený obvod měl stejné parametry jako vzor, popřípadě „o něco lepší“, například aby byl rychlejší, měl nižší spotřebu, větší teplotní toleranci atd.

Pokud měl obvod „druhý zdroj“, popřípadě i víc, používali jej návrháři ráději. Nikdo nechtěl být v situaci, že do zapojení použije obvod, který vyrábí jediný výrobce na světě... Jeho nadřízení se jej okamžitě zeptali: „A co se stane, když zkrachuje? Ne, ne, použijte něco, co vyrábí víc výrobců...!“

Stejný princip tehdy fungoval i u procesorů. První osmibitový mikroprocesor 8008 tak kromě Intelu vyráběl i Siemens (SAB8008), Microsystems International (MF8008) a východoněmecký KME (U808D).

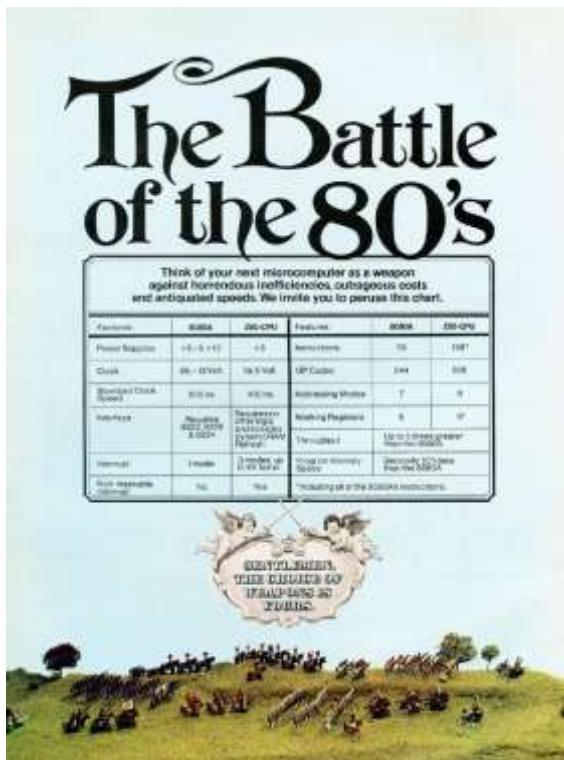
Ekvivalentní procesory vznikaly buď oficiálně – licencováním, nebo trošku černějším způsobem. Nepředpokládám, že Intel prodal licenci na procesor 8080 Tesle. Podle toho, co vím, vznikaly „východní“ mikroprocesory pomocí zpětného inženýrství pokoutně dovezených čipů, studia datasheetů a trošky průmyslové špionáže. Výsledky nebyly někdy stoprocentně shodné, například se vypráví (já nemohu ověřit), že východoevropské ekvivalenty měly občas vyšší spotřebu a horší dynamické parametry, takže například nešly tak snadno přetaktovat, někdy zase naopak byly v některých parametrech lepší.

Každopádně už zmíněný procesor 8080A, vyvinutý Intelem, má co do ekvivalentů velmi pestrou historii. Dodnes známý a existující výrobce AMD vytvořil vlastní klon 8080A pomocí zpětného inženýrství a pod označením 9080A jej začal nabízet v roce 1975. V roce 1976 podepsal AMD s Intelem dohodu a stal se autorizovaným „druhým zdrojem“.

Svoje 8080A vyráběli například Japonci (Mitsubishi M5L8080A, NEC D8080, OKI MSM8080A, Toshiba TMP9080A), Sovětský svaz (580VM80 – jejich značení je legendární), Polsko i ČSSR (MCY7880, resp. MHB8080A), NatSemi, NTE, Siemens, Signetics i Texas Instruments. Dovedete si představit, že by dneska procesory Core i9 vyrábělo kromě Intelu dvacet dalších výrobců a nabízeli ho pod svými značkami?

Intel pak nabídl vylepšenou variantu 8085A. Samozřejmě i tahle varianta měla spoustu „druhých zdrojů“ – AMD AM8085A, NEC D8085A, Mitsubishi M5L8085A, Siemens SAB8085A, Toshiba TMP8085A... V tehdejším SSSR se vyráběl ekvivalent pod označením IM1821VM85A (jasný, ne?)

Ekvivalent vyráběl i výrobce OKI pod označením MSM80C85A – jejich verze měla výrazně nižší spotřebu než originál od Intelu. Čímž se dostáváme k jedné důležité motivaci, a tou je „vylepšit originál“.



## Zlepšovatelé

Odchod Federica Faggina od Intelu a založení Zilolu je notoricky známá historie. Faggin, který se podílel na návrhu 8080, nakonec od Intelu odešel a přidal se k Zilolu, kde navrhl „silně vylepšený procesor 8080“. Jeho cílem bylo navrhnut prosesor, který by byl na úrovni strojového kódu kompatibilní s 8080. Tedy měl stejně registry a stejně instrukce se stejnými operačními kódy. To znamenalo, že existující binární programy pro 8080 nebylo nutné překompilovávat, což usnadnilo vstup jejich Z80 na trh. Navíc v Zilolu snížili počet potřebných napájecích napětí na jedno (+5 V), celý procesor zrychlili, přidali obvody pro automatický refresh dynamických pamětí, přidali druhou sadu registrů, dva indexové registry a spoustu nových instrukcí, které jednak zvýšily ortogonalitu instrukční sady, jednak přinesly zcela nové operace, například blokové přesuny, bloková porovnání či bitové operace.

I procesor Z80 měl spoustu „druhých zdrojů“. Jejich značení je taky pěkná přehlídka kreativity. Ostatně Zilog dnes svou Z80 označuje Z84C00xx (XX udává rychlosť) a podpůrné obvody (PIO, SIO, CTC, DMA), které se blahé paměti označovaly např. jako Z80PIO, jsou dnes Z84C10... Takže se nelekejte, když místo Z80 uvidíte Z84...

Goldstar vyráběl Z8400A, SGS používal taky Z8400, Ates Z80ACPU, Thomson Z84C00, Toshiba TMPZ84C00. Ale třeba Sharp vyráběl totéž pod označením LH0080, ROHM používal BU18400A, NEC značil D780C, Mostek MK3880, Kawasaki KL5C8400, v NDR vyráběli Z80 pod označením U880D (neplést s U808D!), v SSSR dostal klon označení T34VM1 nebo KP1858BM1.

Kromě těchto klonů vznikly odvozeniny. Například Hitachi navrhlo HD64180, což byla vylepšená a rozšířená verze Z80. Vylepšení bylo hlavně v oblasti technologické (CMOS, mikrokód), rozšíření pak představovaly především zabudované periferie a jednotka řízení paměti. Hitachi pak tuhle verzi licencovalo zpět Zilogu, který ji prodával pod označením Z64180, no a po lehkém přepracování u Zilogů se z tohoto čipu stala známá Z180.

Zmiňovaný klon od Kawasaki zase kupříkladu zrychlil provádění některých instrukcí a umožňoval běh až na 33MHz.

Toshiba integrovala některé periferie spolu s jádrem Z80 do jednoho pouzdra s 84 / 100 vývody pod označením Z84013/Z84015 (CMOS verze Z84C13/Z84C15). Dneska tyto čipy vyrábí a dodává jako „second source“ také... Zilog!

Téměř shodná historie se odehrála i v „paralelní větví mikroprocesorového vývoje“, tedy u Motoroly. První procesor 6800 měl rovněž několik „druhých zdrojů“. Historie s Fagginem se u Motoroly opakuje v podobě odchodu některých inženýrů do nově vzniklé firmy MOS Technology, kde vyvinuli procesor 6502. 6502 se vyrábí dodneška a o jeho vylepšenou variantu 65C816, kombinující 8bitové a 16bitové jádro, se postarali vývojáři z WDC, kterou založil, jak jinak, spoluautor původní 6502.

Motorola nabídlá několik variant 6800 (např. se zakomponovanou pamětí), a nakonec vyvinula procesor 6809, který je jednoznačně vrcholem tehdejší osmibitové éry. Tento procesor má mnoho komplexních adresních módů (například „přečti bajt z adresy, která je v registru X, a pak zvyš hodnotu X o 1“), nabízí dva akumulátory A a B (které se dohromady tváří jako 16bitový registr D), dva šestnáctibitové indexové registry X a Y, dva 16bitové ukazatele zásobníku U a S (to by se to implementoval FORTH,

že?), nabízí (podobně jako 6800 nebo 6502) možnost adresovat paměť zkrácenou adresou v nulté stránce – ale tady nejsme omezeni na nultou stránku, protože si pomocí registru DP můžeme nastavit, se kterou stránkou se pracuje. Uvnitř je i hardwarová násobička  $8 \times 8$  bitů... Navíc je sada silně ortogonální (takže se nestává tak často, že by nějaká kombinace operandů „nešla použít“).

Vlastní 6809 vyráběl opět Hitachi, a opět, jako v případě Z80, připravili japonští návrháři i vylepšenou variantu, která se vyrábí pod označením 6309. K vlastnostem 6809 přidává další dva akumulátory E a F, které do hromady tvoří další 16bitový akumulátor W... který spolu s akumulátorem D tvoří 32bitový akumulátor Q. Přibyl i „odkládací“ 16bitový registr V, přibyl „registrov 0“, v němž je vždycky 0 a který zjednodušuje nulování registrů – místo zapsání přímé hodnoty stačí jen přenést hodnotu z tohoto registru, přibyla nové instrukce, blokové operace, instrukce pro dělení a dlouhé násobení, instrukce pro aritmetiku, které výsledek neukládají do akumulátoru, ale do paměti...

## Zdroje nejsou...

Takováto „výroba ekvivalentů“ a „vylepšených ekvivalentů“ kvetla poměrně dlouho. Šestnáctibitové procesory z rodiny x86 měly ze začátku „druhých zdrojů“ dost, nejen známé firmy AMD a Cyrix. Postupem času se od sebe originál a „druhý zdroj“ oddělily, přestaly být kompatibilní výro dově, pak i parametry, a dnes už se jedná o dva odlišné světy s odlišnými chipsety i pouzdry, i když na binární úrovni do jisté míry stále kompatibilní.

U kolegů z Motoroly to platilo ještě pro verzi 68000 – tu vyráběli Hitachi, Mostek, ST, Rockwell, Signetics, Thomson nebo Toshiba. Verzi 68020 a další už jen Motorola. Klon si vyrobil Philips pod označením 68070, ale o jeho kompatibilitě víc neví.

A tak víceméně skončila u procesorů trošičku divoká éra „druhých zdrojů“. Nejčastější situace, kdy víc výrobců nabízí podobné procesory, je dneska ve světě ARMových procesorů, kdy si několik desítek výrobců licencuje „ARM jádro“ (kterých je taky několik verzí) a každý vyrábí vlastní variantu. Navzájem jsou nezaměnitelné, mají jiné parametry, ale základ je stejný a do jisté míry mohou programy pracovat i na jiném „ARMovém“ procesoru.

Podobně se dodnes v portfoliu několika různých výrobců najde různě vylepšené jádro jednočipu 8052. Atmel nebo Dallas vyrábějí své vlastní varianty tohoto jednočipu, které jsou vývodově kompatibilní.

Na jednu stranu se tím zjednodušil výběr platformy a kompatibilita, na druhou stranu jsme ale ochuzeni o různá vylepšení a plody lidské tvořivosti, jako je UB8830D, což byl východoněmecký klon jednočipu Zilog Z8, který měl v ROM z výroby připravený interpret Tiny BASICu.

## Procesor MCS6501

The advertisement is split into two pages. The left page features the headline "the second of a low cost high performance microprocessor family" above the "MCS6501-MCS6502" model names. It highlights "a great microprocessor family 'the software compatibles'". A large box lists "EASIER OF USE!" features: "mc6502 - pin compatible to the MC6800", "mc6502 - minimum external clock circuitry", "mc6502 - eliminates external clock oscillator", "EASIEST DOCUMENTATION TO USE", "SIMPLE, EASY TO FOLLOW INSTRUCTIONS SIMILAR TO POP-11", and "EASIEST TO USE DESIGN OF SYSTEMS". Below this, under "PERFORMANCE", it lists: "HIGH VERSATILITY, PROGRAMMING CAPABILITY", "TWO REAL, INPUT/OUTPUT PORTS", "TWO POWERFUL INSTRUCTIONS", "EASIEST TO EXPAND", "READY (READY FOR SOME MEMORY OR DRIV

The right page features the headline "MOS C502 SAVES MORE MONEY". It includes a large "X" over the text "Eliminates \$10-\$50 Clock Drivers" with the note "Van, 100% mod. v záruce uvedeného času a servisu". Below this, it says "Last Month We Introduced mc6501 for \$29" and "This Month We Added The MCS6502 For \$25". A note at the bottom right says "If you get a quote for the MC6502, say the answer is \$10 less than purchasing direct from us." The MOS logo is at the bottom center, and the text "MOS TECHNOLOGY, INC. A SUBSIDIARY OF AMERICAN DYNACORP CORP." is at the bottom right.

Jeden z nejklasičtějších příběhů z mikroprocesorového booma v 70. letech je ten o MOS Technology na výstavě WESCON 16. září 1975 v San Franciscu. MOS Technology byli ve světě procesorů nováčci. Na výstavu s sebou přivezli dva nové procesory, MCS6501 a MCS6502, a doufali, že by jich s cenou 20 či 25 dolarů mohli pár na výstavě prodat. Pravidla WESCONu ale neumožňovala prodej přímo ve výstavních prostorách, tak bystrý šéf MOS Technology Chuck Peddle nalákal lidi do hotelového pokoje, kde bylo

„pivo zdarma a čip(s)y za 25 dolarů“. V místnosti byly nádoby plné procesorů 6501 a 6502. Měly vzbuzovat dojem, že výroba je v plném proudu. Ve skutečnosti byly vesmírně vadné kusy. Ale to nevadilo, série 6500 se stala velkým hitem, především díky dostupnosti, nízké ceně a prodejem každému (a ne jen „velkým korporátním zákazníkům“). Řada 6500, a speciálně procesor 6501, má zajímavý příběh, který vedl až k jejich slavnému dni na WESCONu.

Příběh začal v Motorola, kde Chuck Peddle, Bill Mensch a někteří další zaměstnanci na počátku 70. let navrhovali procesor MC6800 a periferní obvody. 6800 nebyl špatně navržený procesor, ale byl hodně drahy. Vývojový kit stál přes 300 dolarů. Chuck pracoval hlavně jako systémový architekt, jehož zodpovědností bylo, aby obvody fungovaly bez problémů společně a aby dělaly, co zákazníci očekávají. Proto s budoucími klienty často hovořil, a přitom si všimnul, že mnohé z nich odrazuje jedna věc, totiž cena. S tímto vědomím se snažil postavit levnější verzi 6800 pomocí novějších technologií, které byly k dispozici (jmenovitě využít místo „enhancement mode MOS“, kterým byla vyráběna 6800, technologii „depletion mode NMOS“). Management Motoroly o tom nechtěl ani slyšet, nějaký levný procesor pro masový trh je nezajímá. Takže Chuck, Bill a další lidé, víc než polovina týmu, co pracoval na 6800, odešli.

Skončili ve firmě MOS Technology, které v té době byla z velké části vlastněná společností Allen/Bradley. Právě zde, v MOS Technology pod vedením Chucka Peddla, vznikly procesory 6501 a 6502. 6501 byl pinově kompatibilní s procesorem MC6800, ačkoli měl jinou instrukční sadu. Mohli jste v návrhu beze změn nahradit 6800 procesorem 6501 a jediné, co se muselo změnit, byl software. Použití „depletion mode NMOS“ přineslo zmenšení struktur, takže se na jednu destičku vešlo více čipů. 6502 byl velmi podobný, dokonce používal skoro totožnou sadu masek, s tím rozdílem, že 6502 měl zapojený oscilátor na čipu a jiné rozmístění vývodů. 6501 vyžadoval externí dvoufázový generátor hodin, stejně jako 6800.

6501 nebyl nikdy zamýšlený jako obvod pro masový trh; Chuck často říkal, že to je „dlouhý nos na Motorolu“. Motorola si toho samozřejmě všimla a podala žalobu, což v důsledku znamenalo, že se Allen/Bradley z MOS Technology stáhli, 6501 byl stažen z prodeje, padla pokuta 200.000 dolarů a zároveň výrazně klesla cena procesorů 6800 a dalších. Všeobecně se předpokládá, že si Motorola vynutila redesign 6501 a tak vznikl 6502, ale tak to není. Oba procesory (a s nimi i 6503, 04 a 05) vznikly souběžně a byly dostupné ve stejnou dobu. Motorola si vynutila konec prodeje 6501,

ale ten stejně nikdy nebyl určen k prodeji masám, protože 6502 byl jednodušší a snadněji použitelný v návrhu. 6502 žil dál, i když MOS Technology koupila firma Commodore. 6502 byl použit v mnoha legendárních strojích, jako Apple 1, Commodore 64 (*ten měl upravenou verzi s označením 6510*) a v mnoha dalších. Obvod 6502, ovšem už v CMOS podobě, je několika společnostmi vyráběn dodneška, téměř 40 let po svém debutu s *pivem zdarma*.

## Hitachi HD6309 – jak ze skvělého udělat úžasné

Už jsem psal, že osmibitové procesory nevyráběla vždy jen ta firma, která je vyvinula, ale i jiné. Někdy legálně, na základě licence, někdy pololegálně, až ilegálně – tj. dotyčný výrobce vyvinul ekvivalentní obvod jen na základě reverzního inženýringu a dostupných materiálů.

Výrobci takových „kopií“ a „klonů“ často vylepšili původní proces, takže výsledný čip byl například rychlejší, měl nižší spotřebu, nebo se na křemík podařilo zaintegrovat i nějaké vylepšení. Zmiňoval jsem příklad Hitachi, který vyvinul vlastní vylepšený klon Z80 a licencoval jej zpět Zilogu, jenž ho opět vylepšil a prodával jako verzi Z180.

Stejný osud potkal i procesor 6809 od Motoroly, opět v rukou japonských vývojářů z Hitachi. Hitachi si licencoval od Motoroly výrobu 6809 a vyráběl ekvivalent s označením HD6809. Zároveň jej také přepracovali (např. zavedli místo kombinační logiky mikroprogramy atd.) a přidali některé nové funkce. Takto upravený procesor ale licence neumožňovala prodávat pod označením 6809, takže jeho kód byl HD6309.

Hitachi jej inzerovalo jako „přímou náhradu 6809“, rychlejší a optimalizovanou. Když se podíváte do originálního datasheetu 6309, popisuje jeho vnitřní organizaci stejně jako Motorola 6809. Tedy akumulátory A, B, indexy X, Y, ukazatele U a S, 16bitový registr D... O jakémkoli vylepšení v tomto směru cudně mlčí.

Když náhrada, tak náhrada, takže někteří začali používat 6309 místo originálního 6809 ve svých počítačích. Po čase ale bystřejší hackeři, většinou z komunity tvůrců systému OS-9, začali objevovat drobné rozdíly v chování oproti 6809, zejména při provádění „nedokumentovaných“ instrukcí, tj. operačních kódů, které neměly oficiálně přiřazené žádné instrukce.

Postupně odhalili, že 6309 má oproti svému vzoru velmi významná rozšíření. Informace se šířily mezi japonskými elektronickými nadšenci a v dubnu 1988 vyšel v japonském časopise Oh!FM článek, který shrnoval nedokumentované vlastnosti 6309. Do zbytku světa se tyto informace dostaly až v roce 1991, kdy informace z článku poslal Hirotugu Kakugawa do konference comp.sys.m6809. Díky těmto informacím byl vyvinut už zmíněný systém NitrOS-9, který využíval právě nedokumentované vlastnosti HD6309. (Až později byl NitrOS-9 přepsán tak, aby fungoval i s originálním 6809)

HD6309 nabízí nové registry:

- Dva osmibitové akumulátory, označené E a F (označení je vžité, ale neoficiální – nezapomínejme, že oficiální dokumentace o rozšíření mlčela). Tyto dva akumulátory dávají dohromady šestnáctibitový registr W (podobně jako A a B dávají D).
- Šestnáctibitové virtuální registry D a W tvoří dohromady 32bitový registr Q
- Šestnáctibitový registr V, který nelze použít jako plný akumulátor, pouze pro přesuny mezi registry (např. instrukcí TFR). Jeho zvláštností je, že jeho hodnota zůstane zachovaná i při RESETu.
- Registr 0, použitelný pouze při přesunech mezi registry. Obsahuje vždy nulu a lze ho využít pro nulování jiných registrů.
- Stavový a řídicí registr MD (8 bitů). Tento registr obsahuje dva příznaky: dělení nulou a provedení ilegální instrukce, podle nichž lze rozpoznat, co vedlo k vyvolání výjimky. Zároveň lze dva příznaky nastavit: jedním se určuje, jestli procesor běží v emulovaném módu nebo nativním, druhým lze zajistit, aby se FIRQ choval jako IRQ. Při resetu je nastaven tak, aby se choval jako 6809.

V emulovaném módu pracuje 6309 stejně jako původní 6809, se stejným časováním instrukcí. Samozřejmě lze použít rozšířené možnosti, nové registry apod. V nativním módu trvají některé instrukce kratší dobu a při přerušení se na zásobník ukládají i nové registry E a F (takže v takovém případě je nutné přepsat obsluhu přerušení, pokud se nějak odkazuje na uložená data).

Procesor 6309 zavedl kromě přerušení i „vnitřní přerušení“ (traps), které je vyvoláno v případě načtení ilegální instrukce nebo při dělení nulou. V takovém případě se uloží registry a je vyvolána obslužná rutina na adrese, uložené na FFF0h (u originálního 6809 rezervováno).

Největší inovace přinesl 6309 v oblasti instrukcí. Originální 6809 dovolovala například aritmetické a logické operace pouze mezi akumulátorem a pamětí. 6309 umí všechny tyto operace provést i mezi dvěma registry.

6309 přinesl i instrukce blokového přesunu. Rozšířil totiž původní instrukci TFR (Transfer) o čtyři nové módy, které pracují s pamětí adresovanou registrem: TFM r1+, r2+; TFM r1-, r2-; TFM r1+,r2 a TFM r1, r2+. Registry r1, r2 mohou být D, X, Y, U nebo S, registr W udává, kolik bajtů bude přeneseno. První dvě instrukce fungují jako prostý přesun se stoupajícími/klesajícími adresami (LDIR/LDDR pro znalce assembleru Z80). Druhé dva tvary přenášejí blok paměti do jednoho místa (vhodné např. pro blokový OUT), resp. z jednoho místa do celého bloku paměti (blokový IN).

Násobení  $8 \times 8$  bylo rozšířeno o možnost násobit  $16 \times 16$  bitů ( $Q = W * D$ ). Přibyly instrukce dělení 16 / 8 bitů a 32 / 16 bitů.

Přibyly instrukce pro bitové manipulace s pamětí – AIM, EIM, OIM a TIM, po řadě AND in Memory, EOR in Memory, OR in Memory a TEST in Memory. Kromě nich se objevily i instrukce pro bitové operace mezi pamětí a registrem, které umožňují např. provést AND mezi negovaným třetím bitem nějakého paměťového místa a šestým bitem v registru apod. Tyto instrukce se jmenují BAND, BIAND, BOR, BIOR, BEOR, BIEOR, LDBT a STBT. Výše zmíněný příklad by se zapsal jako „BIAND A, 6, 3, 0x12“ – Bit Inverted AND, registr A, 6. bit, s pamětí na adrese 0x12, třetí bit.

Samozřejmě přibyly i adekvátní instrukce pro uložení nových registrů na zásobník, pro operace s nimi atd.

6309 má vnitřně 16bitovou architekturu. Například instrukce pro prohození obsahu registrů X a Y (16 bitů) tak interně pracuje s šestnáctibitovým TEMP registrem, takže pro výměnu jsou potřeba pouhé tři přesuny, nikoli šest jako u 6809.

Opět připomínám: všechny tyhle informace jsou neoficiální, Hitachi se k nim nikdy nevyjádřil, a jsou výsledkem mravenčí práce spousty vývojářů a hackerů, kteří zkoumali a zjišťovali, jak která instrukce funguje. *Klobouk dolů!*

Procesor 6309 se vyráběl ve verzích s kmitočtem 2MHz (63B09) i 3MHz (63C09) a nabízel vnitřní oscilátor i vnější oscilátor (tato verze měla na konci označení písmeno E).

# Instrukční sada 6502 pod lu- pou

Při pohledu na tabulku instrukčních kódů zjistíte, že většina osmibitových procesorů má v přiřazování kódů k instrukcím poměrně pravidelné vzorce. Je to samozřejmě dáno požadavkem, aby bylo možné instrukce snadno v procesoru dekódrovat. Stačí se podívat u procesoru 8080 na instrukci MOV d,s. Ta má vždy operační kód binárně vyjádřitelný jako 1 0 d d | d s s s, kde ddd a sss jsou tříbitové kódy registrů.

Podobně je tomu i u procesorů 65xx. Podívejte se na tabulku instrukcí procesorů 6502, 65C02 a 65C816 (k tomuto procesoru se ještě dostaneme). Na první pohled může vypadat chaoticky, ale má svou logiku.

Operační kód instrukce si můžeme představit jako osmibitové binární číslo

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK b	ORA (d,X)	cop b	ora d,S	Tsb d	ORA d	ASL d	ora [d]	PHP	ORA #	ASL A	phd	Tsb a	ORA a	ASL a	ora al
1x	BPL r	ORA (d),Y	Ora (d)	ora (d,S),Y	Trb d	ORA d,X	ASL d,X	ora [d],Y	CLC	ORA a,Y	Inc A	tcs	Trb a	ORA a,X	ASL a,X	ora al,X
2x	JSR a	AND (d,X)	jsl al	and d,S	BIT d	AND d	ROL d	and [d]	PLP	AND #	ROL A	pld	BIT a	AND a	ROL a	and al
3x	BMI r	AND (d),Y	And (d)	and (d,S),Y	Bit d,X	AND d,X	ROL d,X	and [d],Y	SEC	AND a,Y	Dec A	tsc	Bit a,X	AND a,X	ROL a,X	and al,X
4x	RTI	EOR (d,X)	wdm	eor d,S	mvp s,d	EOR d	LSR d	eor [d]	PHA	EOR #	LSR A	phk	JMP a	EOR a	LSR a	eor al
5x	BVC r	EOR (d),Y	Eor (d)	eor (d,S),Y	mvn s,d	EOR d,X	LSR d,X	eor [d],Y	CLI	EOR a,Y	Phy	tcd	jmp al	EOR a,X	LSR a,X	eor al,X
6x	RTS	ADC (d,X)	per rl	adc d,S	Stz d	ADC d	ROR d	adc [d]	PLA	ADC #	ROR A	rtl	JMP (a)	ADC a	ROR a	adc al
7x	BVS r	ADC (d),Y	Adc (d)	adc (d,S),Y	Stz d,X	ADC d,X	ROR d,X	adc [d],Y	SEI	ADC a,Y	Ply	tdc	Jmp (a,X)	ADC a,X	ROR a,X	adc al,X
8x	Bra r	STA (d,X)	brl rl	sta d,S	STY d	STA d	STX d	sta [d]	DEY	Bit #	TXA	phb	STY a	STA a	STX a	sta al
9x	BCC r	STA (d),Y	Sta (d)	sta (d,S),Y	STY d,X	STA d,X	STX d,Y	sta [d],Y	TYA	STA a,Y	TXS	txy	Stz a	STA a,X	Stz a,X	sta al,X
Ax	LDY #	LDA (d,X)	LDX #	lda d,S	LDY d	LDA d	LDX d	lda [d]	TAY	LDA #	TAX	plib	LDY a	LDA a	LDX a	lda al
Bx	BCS r	LDA (d),Y	Lda (d)	lda (d,S),Y	LDY d,X	LDA d,X	LDX d,Y	lda [d],Y	CLV	LDA a,Y	TSX	tyx	LDY a,X	LDA a,X	LDX a,Y	lda al,X
Cx	CPY #	CMP (d,X)	rep #	cmp d,S	CPY d	CMP d	DEC d	cmp [d]	INY	CMP #	DEX	wai	CPY a	CMP a	DEC a	cmp al
Dx	BNE r	CMP (d),Y	Cmp (d)	cmp (d,S),Y	pei d	CMP d,X	DEC d,X	cmp [d],Y	CLD	CMP a,Y	Phx	stp	jml (a)	CMP a,X	DEC a,X	cmp al,X
Ex	CPX #	SBC (d,X)	sep #	sbc d,S	CPX d	SBC d	INC d	sbc [d]	INX	SBC #	NOP	xba	CPX a	SBC a	INC a	sbc al
Fx	BEQ r	SBC (d),Y	Sbc (d)	sbc (d,S),Y	pea a	SBC d,X	INC d,X	sbc [d],Y	SED	SBC a,Y	Plx	xce	jsr (a,X)	SBC a,X	INC a,X	sbc al,X

<http://nparker.llx.com/a2/opcodes.html>

<http://nparker.llx.com/a2/opcodes.html>

# Jak fungují nedokumentované instrukce 6502?

Snad každý procesor osmibitové éry v sobě ukryval nějaké překvapení, o kterém se v manuálu nepsalo. Procesor 8085 měl několik šikovných instrukcí, které jsme si představili. Stejně tak to měl i Z80 – tam se před člověkem otevřel velmi široký svět, ve kterém šlo pracovat s polovinami indeksových registrů, posouvat obsah apod.

Naprostým přeborníkem v nedokumentovaných instrukcích je ale procesor 6502. Zatímco výše zmíněné procesory působí dojmem, že „nedokumentované“ někdo opravdu navrhl, zabudoval, ale pak se rozhodli, že o nich nikomu neřeknou, tak u 6502 působí nedokumentované („ilegální“) operační kódy jako směs instrukcí s naprosto nahodilými efekty, od použitelných přes obskurní až k naprosto ujetým.

Použitelná by mohla být (s trohou fantazie) třeba instrukce ANC #imm, která provede operaci AND mezi obsahem registru A a přímým operandem a nastaví hodnotu příznaku C podle nejvyššího bitu výsledku. Trošku dívcejší instrukce jsou třeba LAX (která funguje jako LDA a LDX najednou, tj. vloží operand do registrů A a X) nebo XAA (přenese obsah registru X do registru A a pak provede logický součin [AND] registru A s operandem). Naprosté obskurity jsou instrukce jako TAS nebo SAY.

Kupříkladu taková instrukce TAS. Ta má tvar TAS \$nnnn, Y – tedy operační kód a dva bajty adresy. Jako příklad dejme TAS \$1234, Y:

Instrukce udělá logický součin (AND) obsahu registrů A a X (oba registry ponechá nezměněné) a uloží výsledek do ukazatele zásobníku S.

S výsledkem udělá logický součin s hodnotou \$13 (tj. vyšší byte adresy, zvýšený o 1) a výsledek uloží do paměti na adresu \$1234

Vzbuzuje to několik otázek. Například: Jak na to, proboha, někdo mohl přijít? Nebo: K čemu to je? Nebo: Proč?

Na tu poslední si odpovíme právě v této kapitole. Budu vycházet z famózního článku „How MOS 6502 Illegal Opcodes really work“.

---

*Jak 6502 dekóduje instrukce?*

Na rozdíl od procesorů s mikrokódem, ve kterých je každá instrukce přeložena interně do posloupnosti jednoduchých operací, které jsou provedeny jakýmsi „vnitřním procesorem“, používá 6502 kombinační logiku, jejímž srdcem je PLA – dekódovací paměť. PLA je organizována jako 130 řádků (údajů), každý po 21 bitech. Každý řádek udává operaci (či operace), co se pro daný operační kód (či sadu operačních kódů) provedou v určitém taktu.

Každý údaj lze rozdělit na tři části: ON bity, OFF bity a časování. Zjednodušeně si je můžeme představit takto:



Pozornému čtenáři neuniklo, že nesedí počet bitů (22) s deklarovaným. To je proto, že ve skutečnosti vypadá řádek o něco složitěji – ale k tomu se hned dostanu.

Části ON a OFF určují, jak má vypadat instrukční kód. Pokud má mít nastaven bit 4, bude nastaven bit 4 v části ON. Pokud má mít bity 3 a 2 nulové, budou nastaveny bity 3 a 2 v části OFF. Na hodnotě bitů, které nejsou nastaveny ani v ON, ani v OFF, nezáleží a může být jakákoli. Poslední, co se kontroluje, je takt. V prvním taktu instrukce (po jejím vyzvednutí) je procesor v taktu T1 a s každým dalším se zvětšuje o 1, tj. posouvá doleva v poli „časování“.

Pokud všechny byty, na kterých záleží, odpovídají dané hodnotě, a pokud sedí i číslo taktu, je daný řádek „platný“ a předává se kombinační logice, která podle toho vykoná nějaké jednoduché úkony. Pro jednu instrukci může být platných řádků i více najednou.

## *Skládání instrukcí*

---

Ve skutečnosti je v PLA uloženo pouze šest ON bitů a šest OFF bitů pro bity 2-7. Hodnoty bitů 0 a 1 se kódují trochu jinak, pomocí trojice signálů G1, G2 a G3. G1 platí, pokud je nastavený bit 0, G2 pro nastavený bit 1 a G3 pro oba bity nulové. Vidíte, že logika není úplná, že chybí ošetření stavu pro oba bity nastavené. A je to opravdu tak, při pohledu do tabulky instrukcí vidíme, že instrukce, které mají nastavené oba nejnižší bity (tj. končí na x3, x7, xB a xF) jsou „ilegální“. Ve skutečnosti takové instrukce spustí řádky s G1 i s G2 a fungují tak jako kombinace dvou předchozích instrukcí.

(Když se ponoříte do výpisu PLA, zjistíte, že jsou bity v naprosto odlišném pořadí, proto opakuju: výše zmíněné je pouze ilustrační!)

Zjednodušeně řečeno: pokud má instrukce oba nejnižší bity nastavené, chová se, díky neúplnému dekodéru, jako dvě instrukce, G1 a G2. Například instrukce s kódem \$AF se chová jako instrukce s kódem \$AE – LDX i s kódem \$AD – LDA (nikoli \$AC, ta má oba bity nulové a je správně dekódována jako G3). U prvních dvou taktů to nevadí, v nich se pouze čte absolutní adresa, ve třetím taktu se obsah této adresy čte do registru. Do kterejho? To se nastavuje v prvním taktu. Pro G1 se aktivuje signál, uvolňující zápis do registru A, pro G2 zápis do registru X. V našem případě se tedy uvolní oba, a hodnota se zapíše do obou.

Obdobně kombinace STA a STX vytvoří složenou instrukci SAX, která do paměti uloží hodnoty obou registrů A a X najednou – totiž výsledek AND obou hodnot (tipuji, že jsou interně použity otevřené kolektory, které vytvoří „montážní AND“).

Takto lze vystopovat mnoho „ilegálních instrukcí“. Dokonce i TAS...

---

## *Kill 'em!*

V PLA je zakódováno nejen to, kolik taktů která instrukce trvá a kde bere operandy, ale i načtení kódu další instrukce v posledním taktu. Což s sebou nese další zajímavý jev.

Některé kódy nedělají nic (NOP). Některé NOPy ještě předtím načtou jeden nebo dva bajty. No a některé instrukce prostě zaseknou celý procesor

(většina kódů, které končí dvojkou). Nejpravděpodobnější vysvětlení je, že se u těchto kódů v PLA nenajde právě to načtení další instrukce, které mj. vynuluje počítadlo taktů, a instrukce se dostane do smrtícího „osmého taktu“, pro který už neexistuje žádný záznam v PLA. A protože se požadavky na přerušení řeší až na konci instrukce, tj. v okamžiku, kdy se nuluje počítadlo taktů, tak ani přerušení nenastane. Procesor se zkrátka „ocitne mimo šachovnici“ a vzpamatuje ho až RESET.

K dalšímu studiu doporučuju články:

- Decode ROM
- How MOS 6502 Illegal Opcodes really work
- PLA dump
- 6502 opcodes
- 6502 extra opcodes

# Procesor Hitachi HD6309



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

**Podpořte její vznik?**

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na <https://www.osmibity.cz/addons.html>

# Procesor 65C816 – šestnáctibitový 6502

<http://mdfs.net/Docs/Comp/65816/intro.htm>

# Dodatky

## Monitor c'mon pro OMEN Bravo

Monitor c'mon je jednoduchý ovládací program, který používá řada konstrukcí s procesorem 6502. Použil jsem jej proto i já, ačkoli se jeho ovládání liší od standardního monitoru OMEN.

Nejdůležitější rozdíl je v tom, že se příkazy nezadávají jako příkaz a parametry, tedy například M2000 pro výpis paměti od adresy 2000h, ale obráceně, tedy nejprve adresa, které se příkaz týká, a teprve pak příkaz.

Pro vypsání obsahu paměti slouží příkaz X (eXamine). Monitor nerozlišuje velká a malá písmena. Paměť od adresy 2000h vypíšete tedy obráceným postupem:

```
2000x
```

Pro ukládání hodnot do paměti slouží příkaz @. Hodnot můžeme zapsat víc a oddělit je čárkou:

```
2000@10,20,30,DE,AD,BE,EF
```

Program spustíme příkazem G:

```
2000G
```

Ve verzi pro Bravo jsem použil c'mon se zapnutým pluginem pro krokování. Pokud máte program uložený v RAM, můžete použít příkaz \$ pro provedení jednoduchého kroku. Nejprve pomocí @ nastavíte adresu, a pak pomocí \$ vykonáte jednu instrukci:

```
2000@
$
```

a tak dále... Monitor po každém kroku vypíše obsah registrů PC, A, S, X, Y a P (ten vypíše hexadecimálně i binárně).

Pro snazší používání jsem doplnil i funkci nahrávání souborů HEX. Stačí pustit po sériové lince soubor HEX. Monitor podle úvodní dvojtečky pochopí, že následuje řádek HEX souboru, a až do konce řádku s příchozími znaky nakládá tak, že z nich načte adresu, délku řádku a data.

Dokumentaci k c'mon pro šestnáctibitový 65C816 najeznete na adrese <https://biged.github.io/6502-website-archives/lowkey.comuf.com/cmon.htm> nebo v repozitáři Bravo na GitHubu.

---



Chcete přispět ke vzniku této publikace?

Kniha vzniká ve volném čase, bez smlouvy s vydavatelem, a je dostupná zdarma pod otevřenou licencí.

**Podpoříte její vznik?**

Autor ocení jakýkoli příspěvek, ideálně pomocí PayPal. Více informací na <https://www.osmibity.cz/addons.html>

