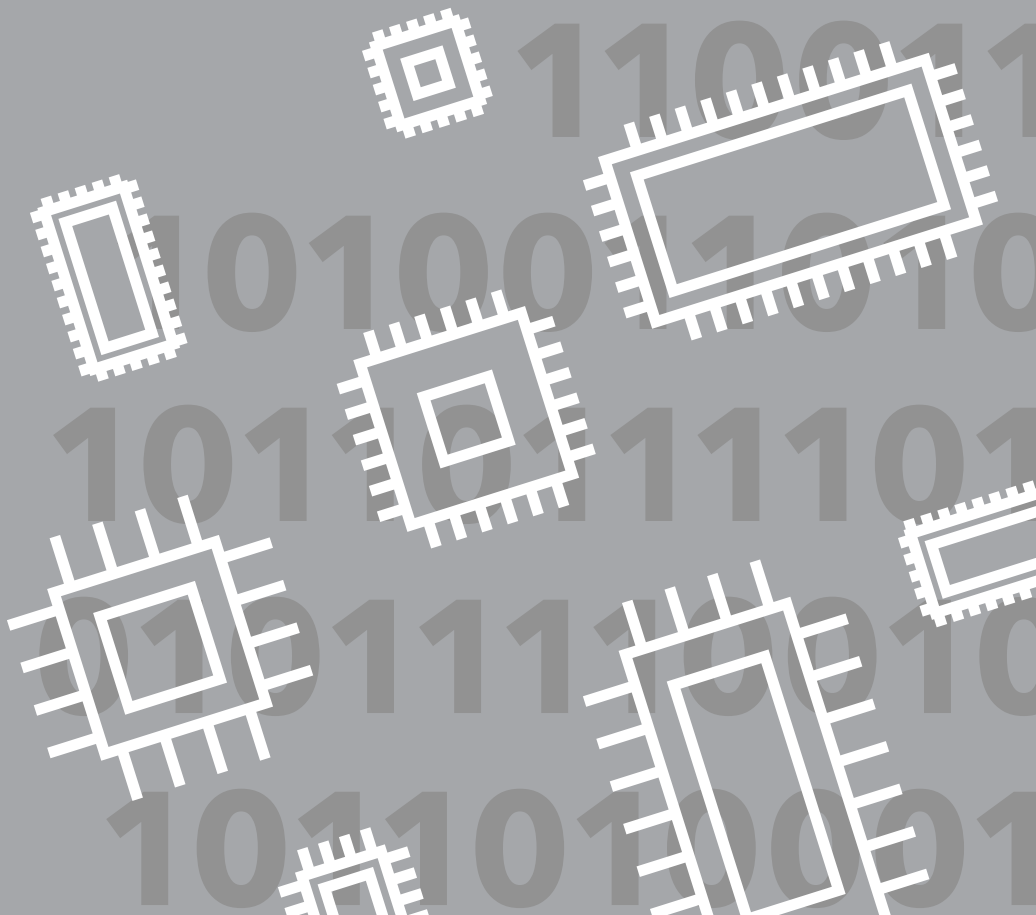
Martin Malý

# Data, čipy, procesory

## Vlastní integrované obvody na koleni



Edice CZ.NIC

**DATA, ČIPY, PROCESORY**
**Vlastní integrované obvody na koleni**

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2020
Kniha vyšla jako 25. publikace v Edici CZ.NIC.
ISBN 978-80-88168-56-0



CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-88168-56-0

— Martin Malý

# Data, čipy, procesory
## Vlastní integrované obvody na koleni

— Edice CZ.NIC

# Poděkování

## Poděkování

Děkuju všem lidem, kteří mi při psaní fandili. Všem čtenářům předchozích knih, co mi psali, že s nimi je začala elektronika zase bavit. Všem lidem, kteří mi psali, že se těší na pokračování. Všem těm, kteří četli rukopis a přispěli radami a názory. Všem těm, kteří mi dali ve chvílích pochybností podporou najevo, že to není úplně zbytečná práce, že to někoho zajímá a že nepíšu do zdi.

Speciální díky patří nadšencům, kteří podpořili vznik knihy na Patreonu i jinde: Josef Adamčík, Stanislav Jurný, Michal Kočer, Martin Ludik, Karel Nenička, Filip Novák, Dan Tománek, Radomír Vávra, Kamil Zmeškal – a určitě jsem na někoho zapomněl, za což se omlouvám.

Velké poděkování patří vydavateli a všem lidem z Edice CZ.NIC, kteří z rukopisu udělali knihu. Už potřetí!

A v neposlední řadě patří poděkování mé tolerantní partnerce Míše.

Díky, díky, díky!

— Poděkování

# Předmluva vydavatele

## Předmluva vydavatele

Vážení čtenáři,

dostává se vám do ruky třetí kniha od Martina Malého, tentokrát na téma programovatelná hradlová pole (FPGA, anglicky Field Programmable Gate Array). Na předchozí dvě navazuje jen velmi volně, ovšem hodí se mít pročtené „Hradla, volty, jednočipy". Předchozí knihy vám pomohou zasadit nové informace do správného kontextu, ale jelikož je téma dost odlišné, neztratíte se, ani pokud je neznáte.

Vždycky jsem obdivoval lidi, kteří hardware rozumí a ví, co se uvnitř děje a jak. Já sám jsem se nikdy nedostal o moc dál, než jsou základy ze středoškolské fyziky. Vývoj software mě pohltil daleko víc a vydal jsem se touto cestou. I proto je pro mne téma FPGA velmi lákavé. Je to možnost, jak vyžít své zkušenosti a schopnosti úplně jiným způsobem – ponořit se do návrhu složitého hardware, i když své hardware znalé kolegy běžně děsím tím, co si doma bastlím a jak.

Než knihu začnete číst, měl bych jedno varování pramenící z mé osobní zkušenosti s touto knihou. Není snadné ji dočíst. Nebudete ani ve třetině, když vás přepadne neodvratné nutkání pořídit si nějaký ten dev kit a příklady si zkoušet naživo. Přeci jen číst si o tom a opravdu to dělat, jsou dvě různé věci. Doporučuji však pokusit se nutkání odolat. Vím, není to snadné. Ale i když už začátek zní velmi lákavě, vyplatí se počkat si na pozdější kapitoly, kdy se třeba dozvíte, jak generovat grafický výstup, připojit SD kartu nebo kde najít již připravené moduly. Sčítání bitů zní zajímavě, ale teprv když zařízení dělá něco samo o sobě (bez debuggeru), tak to má to správné kouzlo. A během čtení se vám budou postupně odkrývat nové a nové možnosti, co s FPGA dělat, a nápady, co s FPGA podniknout, budou jen a jen přibývat. A určitě některé z nich ovlivní i výsledný výběr dev kitu.

Ohledně FPGA najdete na Internetu spoustu informací, ale není úplně snadné rozmyslet si, kde a čím začít. Tato kniha vás však velmi poutavou formou se světem FPGA snadno seznámí, navnadí vás a dá vám užitečné stavební kameny do začátku. Pak už zbývá si jen pořídit dev kit, vše si prakticky vyzkoušet a začít tvořit.

Přeji příjemné čtení a mnoho zajímavých pokusů s FPGA.

Michal Hrušecký, CZ.NIC

# Předmluva

## Předmluva

Psal se rok 2005 a já se probíral příspěvky v internetové konferenci elektroniků, elektrotechniků a vůbec elektrošotoušů a bastlířů. Většina příspěvků se točila okolo tehdy populárního jednočipu PIC16C84, popřípadě kolem toho, jaké vybavení je pro dílnu dostatečné, a najednou do diskuse vstoupil *opravdový amatér.* Člověk, pro kterého byla mikroelektronika kouzelný svět, který s nadšením objevoval a se kterým se seznamoval.

Na začátku příspěvku se omluvil za to, že se vůbec dovoluje na něco zeptat (ano, to patřívalo k bontonu... *omluvte mě, vy moudří, že mám hloupý začátečnický dotaz),* a pak položil skvělou otázku: Jestli by mu někdo nemohl poradit, jak by si mohl splnit svůj sen, totiž **navrhnout si vlastní mikroprocesor.**

A já se opět, jak mávnutím kouzelného proutku, ocitl v roce 1984 v okresní knihovně, v oddělení techniky, a držel jsem v ruce knihu „Polovodičové paměti a jejich použití". Hned v úvodu autor popisoval, jak se vytvářejí tranzistory MOS a jak pracují. Bylo to tak jasné a pochopitelné, že jsem věřil – opravdu jsem tomu věřil! – že je možné si takové tranzistory dělat doma na koleni. Však křemík je všude, ty donory a akceptory by se taky někde sehnat daly, to přeci musí jít! A věřil jsem tomu, že jednou, jednoho dne, si ve sklepě v Petriho misce stvořím vlastní polovodič! Inu, byl rok 1984, bylo mi 11 let a připadalo mi snazší si udělat vlastní integrované obvody, než doufat, že si je jednou koupím v Elektře na rohu.

Nota bene když jsem v ruce držel knihu, kde to všechno bylo popsané. Jak se z tranzistorů poskládají hradla, z hradel klopné obvody, multiplexory, matice paměťových buněk... prostě všechno. I vlastní mikroprocesor bych zvládnul, určitě!

Úplně jsem cítil atmosféru té knihovny a svou dětskou radost, když jsem si do sešitu obkresloval tranzistory, křemíkové struktury a obvody a navrhoval jsem si vlastní komponenty. Tužkou, na papíře... Moc se mi líbilo, že si někdo takový sen udržel i po dvaceti letech a statečně se zeptal: „chtěl bych si navrhnout vlastní mikroprocesor, jak na to?"

Nepřekvapivě dostal *neskutečnej kartáč* od osazenstva konference, plus mínus ve stylu „my tu řešíme reálné problémy a na takovéhle bláznivé fantazie tady nejsme zvědaví, kšá!" *Vlastní procesor? To nejde, na to nikdy mít nebudeš, to nikdy nezvládneš, to ti žádná fabrika nikdy nevyrobí, to nikdy nikdo nebude používat, tak s tím neotravuj.*

A po zhruba dvaceti odpovědích v tomto stylu přišel jeden z členů té konference a tazatele nezadupal, ale naopak ho povzbudil. Ať si z těch řečí kolem nic nedělá, ať to klidně zkusí, protože se u toho naučí o elektronice mnohem víc než všichni ostatní účastníci té diskuse dohromady, a ať se nebojí, že to nevyjde, protože i tak získá spoustu neocenitelných znalostí. No a nakonec dodal, že nejjednodušší bude podívat se na obvody CPLD nebo FPGA a naučit se nějaký HDL,

jazyk, kterým je možné ty obvody programovat.

Jak to s tazatelem dopadlo a jestli si vlastní procesor někdy navrhl, to netuším. Já si jen pamatuju, že jsem si kamsi v hlavě udělal poznámku: FPGA, HDL, *zajímavé.* A pak to na dlouhé roky vytěsnil, protože to bylo ve škatulce „drahé, nedostupné, není na hraní" - a já chci elektroniku hlavně na hraní.

Není to tak dlouho, tak sedm let, kdy se najednou objevily levné kity s FPGA, dostupné i pro nás, bastlíře, a začaly se objevovat první nesmělé pokusy a první konstrukce. A tehdy jsem si i já koupil svůj první kit, asi za tisícovku, a po několika týdnech experimentů jsem stvořil funkční repliku mikropočítače PMI-80.

Dnes jsou FPGA ještě dostupnější, a ti z vás, co už vymačkali maximum ze svých Arduin, BluePill a jiných Raspberry, se možná začínají poohlížet právě po těchto obvodech. Nedivím se – pověst, která je předchází, je zajímavá. *Obvod, který může být čímkoli, co zvládnete nadefinovat,* no není to sen?

Možná se trochu bojíte, možná máte v hlavě nějaký vnitřní majáček, který vám říká, že *to je složité, nezvládnete to, na nic to nebude...* Kašlete na majáček! Vážně! Pusťte se do toho.

Kniha, kterou právě držíte v ruce a kterou si za chvíli koupíte, je přesně to, co potřebujete, aby se z vás, z člověka, *co by to rád zkusil,* stal člověk, který si to zkusil – a možná ho to chytlo a bude pokračovat! Ukážeme si, co jsou vlastně FPGA, jaké možnosti coby amatér máte, a pak se naučíme jeden z univerzálních jazyků pro popis elektroniky, totiž VHLD. Zabrousíme i do Verilogu, naučíme se obvody navrhovat, testovat, simulovat, naučíme se, jak se ve VHDL zapisují základní konstrukční prvky, jak se skládají dohromady, jak se vytvářejí obvody pomocí popisu jejich chování, připravíme si sadu užitečných elementů pro vlastní pokusy, a pak si ukážeme nejen to, jak ve FPGA vytvoříte celý počítač, ale i to, jak si uděláte vlastní mikroprocesor.

Kniha není ani technická příručka, ani učebnice. Na to je příliš hovorová a příliš populární. Nenahradí vám vysokoškolská skripta a po jejím přečtení asi nebudete připraveni nastoupit do vývojové laboratoře a živit se jako konstruktér s FPGA obvody. To ani není její cíl. Její cíl je jiný: ukázat vám zajímavý svět obvodů FPGA a uživatelsky definované elektroniky, což je dnešní „hi-tech" oblast, zbavit vás ostychu a strachu, že *tomu nebudete rozumět,* a povzbudit ve vás chuť zkoušet a vymýšlet nové věci.

I kdyby to mělo být něco neužitečného a nepraktického.

Pojďme na to!

*Důležité post scriptum: K této knize je dostupný web* ***https://datacipy.cz/****, kde kromě tipů na další čtení a užitečných odkazů najdete i zdrojové kódy všech příkladů z knihy, včetně testů, a také kódy, které byly příliš dlouhé a do knihy se nevešly.*

# Obsah

**Poděkování** **7**

**Předmluva vydavatele** **11**

**Předmluva** **15**

**1 FPGA? Co, prosím?** **25**
1.1 Programovatelné obvody 25
1.2 Jaké FPGA? 30
1.3 Jaký kit vybrat? 34

**2 Základy VHDL** **49**
2.1 Proč se učit VHDL? 49
2.2 Než začneme... 49
2.3 Úplné základy a nezbytná teorie 50
2.4 Hello world! 53
2.5 LUT 58
2.6 Testování 58
2.7 Komponenty a signály 68
2.8 Bit sem, bit tam... 79
2.9 Typy, operátory a atributy 88
2.10 Proces 98
2.11 Hodinové signály a čas 109
2.12 Klopné obvody, registry a další… 114
2.13 Funkce, procedury, balíčky 129
2.14 VHDL 2008 140

**3 Podrobněji o FPGA** **145**
3.1 Jak FPGA pracují? 145
3.2 Piny a jejich přiřazení 145
3.3 Hodinové signály 147
3.4 Nahrávání konfigurace do kitu EP2C5 147

**4 Analogový výstup** **157**
4.1 PWM 157
4.2 Pokus: FPGA siréna 164

**5 Paměti** **173**
5.1 Obousměrná sběrnice 173
5.2 Paměti RAM (RWM) 174
5.3 Paměť ROM 180

5.4 IP: Hotové paměti 180
5.5 Pokus: Melodický zvonek 183

**6 Čítače 187**
6.1 Binární čítače 187
6.2 Speciální čítače 190
6.3 Problém s přenosem 192

**7 Automaty 195**
7.1 Konečné automaty 195
7.2 UART 197

**8 Hodinové domény 207**
8.1 Hodinové domény 207
8.2 UART, druhý díl – přijímač 215

**9 Generátor (pseudo)náhodných čísel 223**
9.1 LFSR 224

**10 IP, OpenCores a hardware s FPGA 231**
10.1 Multicomp 233
10.2 MiST 234
10.3 ZX Spectrum Next 235
10.4 Gameduino 236

**11 OMEN Alpha, tentokrát ve FPGA 239**

**12 Generování VGA videosignálu 249**
12.1 VGA teoreticky 249
12.2 Synchronizace 250
12.3 R, G, B 252
12.4 PLL 252
12.5 Kalkulačka! 254
12.6 Jednoduchý obrazec 255

**13 Užitečné obvody 261**
13.1 Dekodér pro sedmisegmentovky 261
13.2 Multiplexní buzení sedmisegmentového displeje 263
13.3 Generická dělička kmitočtu 265
13.4 Generátor úvodního signálu RESET 266
13.5 Debouncer 267

13.6 Sériové rozhraní SPI 269
13.7 Rozhraní I²C 276
13.8 Připojení SD karty 279
13.9 Generátor parity 281
13.10 Připojení PS/2 282
13.11 SDRAM 285
13.12 HDMI 290

**14 Vlastní mikroprocesor 295**
14.1 Architektura mikroprocesoru 296
14.2 Přípravné práce 297
14.3 Mikroprocesor MHRD 305

**15 Stručný úvod do Verilogu 317**
15.1 Syntaktické základy Verilogu 319
15.2 Datové typy 320
15.3 Operátory 322
15.4 Moduly 322
15.5 Porty 323
15.6 Příkaz assign 324
15.7 Blok always 326
15.8 Testování – blok initial 329
15.9 Stručné shrnutí základů Verilogu 332
15.10 Parametrizace modulů 333
15.11 Blokové instrukce 335
15.12 A dál? 337

**16 Verilog prakticky 341**
16.1 FORTH a procesor J1 341
16.2 Implementace procesoru J1 ve Verilogu 345
16.3 Verilog vs VHDL 352

**17 Doslov 357**

**18 Příloha: Kit EP2C5T144 361**
18.1 Mapa obsazených pinů 361

**19 Příloha: Kit OMDAZZ 365**

**20 Příloha: VHDL v kostce 369**
20.1 Operátory 369

20.2 Atributy 370
20.3 Deklarace 372
20.4 Rozhodování (resolution) 379
20.5 Sekvenční příkazy 381
20.6 Konkurenční příkazy 386

# 1 FPGA? Co, prosím?

# 1 FPGA? Co, prosím?

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

No dobře, přeháním, ani sestavování kombinačních obvodů není smrtící, ale představte si takový dekodér, jaký jsme používali v knize Porty, bajty, osmibity.

(Pokud jste tuto knihu nečetli, tak vám prozradím, že jsme většinou dekódovali 16 bitů adresové sběrnice tak, aby byl adresován jeden ze dvou paměťových obvodů, popřípadě jsme dekódovali tři bity adresy na 8 různých signálů pro výběr obvodů.)

Teď si jej představte o něco složitější. Představte si jemnější škálování adresního prostoru, třeba po 1 kB blocích, to máte šest adresních vstupů, a představte si složitější paměťovou mapu, kde třeba prvních 8 kB je RAM, pak 8 kB ROM, pak 16 kB zase RAM, 1 kB prostoru pro periferie, ten se patnáctkrát zrcadlí, posledních 16 kB je zase ROM, ale stránkovaná...

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

Naštěstí moderní elektronika udělala od sedmdesátých let obrovský skok kupředu, a během uplynulé dekády se i ty nejmodernější obvody staly dostupné běžným smrtelníkům – tedy lidem, jako jsme my. Dnes máme možnost navrhnout si celé systémy, které vyžadovaly desítky či stovky integrovaných obvodů, pomocí jazyků pro formalizovaný popis logických obvodů, v počítači vše nasimulovat, a nakonec nahrát do „prázdného obvodu“, který se tak promění v cokoli, co chceme.

A právě o tom je celá kniha, kterou právě držíte v ruce!

## 1.1 Programovatelné obvody

### PROM

V letech dávno minulých se podobné problémy, pokud jste na to měli příslušné vybavení, řešily pomocí pamětí 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.

### Zákaznické obvody

Někteří výrobci čipů nabízeli „prázdné logické obvody“, ovšem s tím, že jejich konfiguraci si zadal zákazník. Takový obvod nabízela třeba i Tesla, ale nejznámější v našich končinách bude výrobce Ferranti a jeho zákaznický obvod ULA, použitý v ZX Spectru. Ve skutečnosti šlo o předchůdce pozdějších obvodů CPLD, kde se konfigurace neukládala do paměti, ale byla z výroby natvrdo „vypálena“ do křemíku. Samozřejmě pro kusovou výrobu byly takové obvody nedostupné, ale pokud jste jich chtěli odebrat tisícové série, měli jste možnost.

### 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 pouzdru. 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 napevno přiřazené určité vnitřní bloky ke konkrétním pinům.

Hlavní rozdíl je ale ten, že CPLD obsahují řádově víc 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. Typ s označením 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 jazyky, řazené do rodiny jazyků HDL (Hardware Definition Language), typicky VHDL nebo Verilog. V těchto jazycích (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 dále. 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, ...

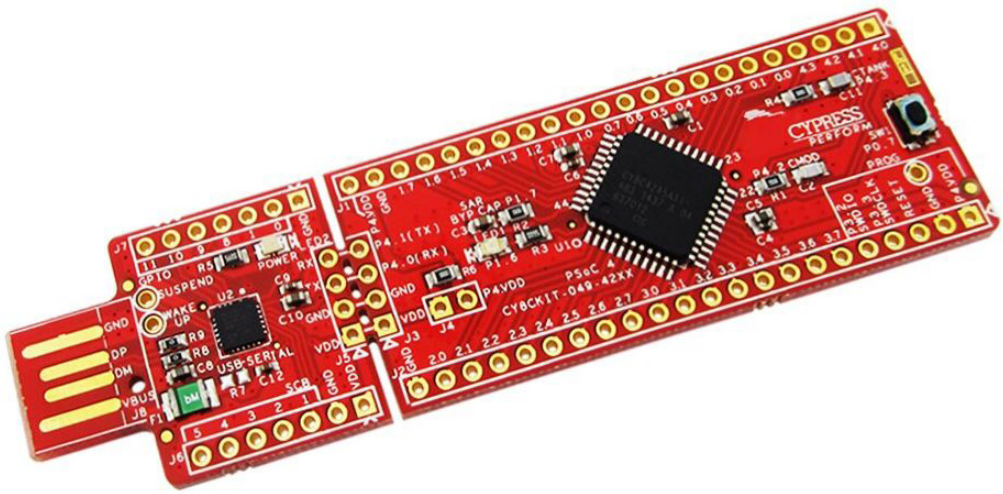
Na rozdíl od obvodů CPLD mívají obvody FPGA svou konfiguraci uloženou nikoli ve vnitřní paměti, ale v paměti vnější, nejčastěji v podobě sériové FLASH. Po startu systému se z této paměti načte konfigurace do FPGA.

### SoC

Zajímavý koncept představují obvody SoC (System-on-a-Chip), které v sobě kombinují mikroprocesorové jádro, paměť, standardní periferie a programovatelnou logiku. Vývojáři tak mohou i některé specializované funkce syntetizovat přímo v obvodu, bez nutnosti vytvářet speciální zapojení.

Příkladem mohou být obvody PSoC od Cypress (nyní Infineon). Rozšiřují běžné mikrokontroléry s jádrem ARM o univerzální digitální bloky UDB. Ty můžete naprogramovat (či přesněji konfigurovat) sami, a to buď v editoru komponent, v grafickém editoru stavových strojů, nebo pomocí jazyka Verilog.

Vývojové kity jsou levné a dostupné, například kit CY8CKIT-049-42XX s cenou do deseti dolarů obsahuje čip CY8C4245AXI-483. Ten nabízí procesor ARM Cortex-M8 s maximální frekvencí 48 MHz, 32 kB FLASH, 4 kB RAM, A/D převodník, operační zesilovače, komparátory, PWM a 4 programovatelné logické bloky. U některých konstrukcí může právě programovatelná logika nahradit buď velkou část výkonu, nebo vlastní konstrukci podpůrných obvodů.




## K čemu mi je FPGA?

Přesně! *To je vážná průmyslová věc, to není hračka pro bastliče!* Jako 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 přesto si můžete 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 – a nebojme se to říct: *bastlení* – 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?

V obvodu FPGA můžete klidně vytvořit hned několik procesorů. Klidně i procesorovou matici. Transputer. Paralelní výpočty tak budou probíhat opravdu paralelně, podobně jako ve vaší grafické kartě. Můžete implementovat klidně neuronovou síť a trénovat ji, na co potřebujete. Díky tomu, že její jednotlivé neurony budou skutečně obvody a ne jen datová struktura v paměti počítače, budou pracovat opravdu paralelně – a opravdu rychle!

Nadšenci do kryptografie a kryptoměn zase mohou FPGA využít jako masivní paralelní počítadla hashů. Rychlost takových počítačů není omezena frekvencí procesoru, ale jen zpožděním při šíření signálu logickými prvky.

FPGA je zkrátka něco jako obří krabice Lega. Ale opravdu obří. Můžete si z ní poskládat téměř cokoli, ale musíte být schopni to poskládat ze základních součástek. A právě v této knize se to společně naučíme.

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

## 1.2 Jaké FPGA?

Pro amatérské použití se moc nehodí nejnovější a nejvýkonnější obvody. Jejich schopnosti jsou daleko před potřebami amatérské praxe a jejich cena vysoko nad možnostmi amatérské peněženky. Ale i v té spodní, dostupné části spektra nalezneme dostatek obvodů pro konstrukci velmi zajímavých zařízení. Cílem této kapitoly 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 (Alteru před nedávnem koupil Intel). 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?

Dva největší výrobci FPGA. 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, resp. Intelu, pak řadu Cyclone, konkrétně Cyclone II a Cyclone IV.

### Typy, řady a generace FPGA

Značení řad FPGA je na první pohled trošku nepřehledné, ale logiku má. Vezměme si jako příklad výrobce Xilinx a jeho obvody FPGA. Xilinx nabízí FPGA s různými názvy a číselnými označeními. Obecně se dá říct, že číselné označení udává „generaci" – Spartan 3, Spartan 4, ... Spartan 7 jsou postupně výkonnější a výkonnější obvody, vytvořené s novými technologiemi (Spartan 6 například je vytvořen 45nm technologií, sedmá generace pak 28nm, generace UltraScale má technologii 20 nm, UltraScale+ používá 16 nm). V prvních generacích šlo o Spar-

tany, později přibyly další typy, jako Artix, Kintex a Virtex. Spartan jsou levné a nenáročné obvody, Virtex ty nejdražší, nejrychlejší a nejlépe vybavené. Každý obvod má několik různých mutací, podle počtu vývodů nebo logických jednotek uvnitř.

V sedmé generaci přišly obvody Spartan 7, Artix 7, Kintex 7 a Virtex 7, všechny vyrobené technologií 28 nm, a každý v několika verzích.

**Spartan** 7 se vyrábí jako XC7S6, XC7S15, XC7S25, XC7S50, XC7S75 a XC7S100. XC je obecné označení FPGA od Xilinxu, 7 označuje generaci, písmeno S naznačuje, že jde o Spartan, a poslední číslo udává počet logických buněk (v tisících). Čím větší obvod, tím víc vývodů (S6 jich má 100, S100 až 400), tím víc jednotek PLL, tím víc integrovaných paměťových bloků (20 kB až 480 kB) atd.

**Artix** stejné generace nabízí obvody XC7A12, A15, ... až XC7A200. Označení je analogické předchozímu, písmeno A udává typ Artix, poslední číslo pak opět hrubý počet logických buněk v tisících (12800 až 215360). Opět platí přímá úměra mezi počtem buněk, počtem vývodů, velikostí paměti (80 kB až 1,4 MB), DSP jednotek atd. Artix ale nabízí i moduly pro PCIe nebo rychlé transceivery.

**Kintex** 7 se skládá z obvodů XC7K70, K160, K325, ... XC7K480. Logika číslování je opět tatáž, platí i přímá úměra mezi velikostí a dalšími parametry. Dá se říct, že Kintex má vybavení jako Artix, jen má všeho o trochu víc a něco má vylepšené. Moduly DSP má vylepšené, modul PCIe je pro verzi 2 atd.

**Virtex** 7 si můžete dopřát v nejmenší variantě XC7V330 (326400 logických buněk, 3000 kB RAM) či v té největší XC7V1140 (1,1 milionu logických buněk, 7 MB RAM), některé modely mají i transceivery schopné přenosu rychlostí 28 Gbps, kromě PCIe verze 2 se objevují i PCIe verze 3...

V době psaní knihy je sedmá generace už překonána generacemi UltraScale a UltraScale+, které přinášejí opět větší množství rychlejších modulů. Řádově miliony logických buněk, megabajty RAM, stovky transceiverů, tisíce DSP...

U Intelu převzali řady, které vyráběla Altera, a pokračují v nich. Potkáváme se tedy s obvody z řad MAX, Cyclone, Arria, Stratix a Agilex.

Cyclone, uvedená na trh v roce 2002, má generace II, III, IV, V a 10 (ano, po Cyclone V přišla generace Cyclone 10). Kromě první a druhé generace jsou všechny stále doporučené pro používání.

Téměř stejné generace jsou u typu Stratix: III, IV, V a 10.

MAX je k dispozici jako generace II, V a 10. Generace II a V představují nikoli FPGA, ale CPLD – jednodušší předchůdce FPGA. Za FPGA výrobce označuje až řadu MAX 10.

„Desátá" generace u Intelu / Altery odpovídá zhruba „sedmé" generaci u Xilinxu. Najdete různé variace na MAX 10, Cyclone 10, Arria 10 a Stratix 10. MAX jsou jednočipová FPGA s integrovanou konfigurační pamětí a spolu s Cyclone tvoří „low end". Obvody Arria a Stratix jsou už z kategorie „System on a Chip" (SoC), protože obsahují výkonné procesory ARM Cortex. Stratix je spíše zaměřený na výpočetní výkon, zatímco Arria na komunikaci.

### Co je vhodné pro začátečníka?

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 jiné obvody pro programování přes JTAG. Jazyky naštěstí můžete použít u obou stejné.

Neexistuje obecná rada, jestli Xilinx nebo Altera. Já dlouho upřednostňoval Xilinx, teď mi připadají kity s FPGA od Altery 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 / Intel – 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, nemluvě o tom, že starší Quartus v edici zdarma najdete na webu s menšími obtížemi než starší ISE WebPack a nové IDE Xilinx Vivado podporuje jen řady 7 a UltraScale, takže třeba pro kit s Virtexem 4 nebo se Spartanem 3 musíte hledat staré ISE, registrovat se a žádat o „licenci zdarma".*

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

### A co Lattice?

Ano, jsou i další výrobci, jako například Lattice se svými obvody Mach, iCE nebo ECP5 / ECP3 / ECP2 / XP. I tento výrobce nabízí vývojové prostředí zdarma, včetně emulátoru, a až na některé detaily se práce s těmito obvody neliší od práce s FPGA od Xilinxu či Altery (Intelu). Někdy ale může být těžší sehnat vhodné vývojové desky či programátory.

### 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 podobný jazyku Pascal, Verilog zase připomíná C. 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 budete potřebovat?

#### 1. Kit

Doporučuju 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.

*V ČR má tento kit v nabídce například e-shop HW Kitchen: https://hwkitchen.cz/*

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. Nyní, po odkoupení Altery společností Intel, bylo vývojové prostředí přejmenováno na Quartus Prime, a ve verzi Lite je zdarma.

https://fpgasoftware.intel.com/13.1/?edition=web

#### 4. Znalost VHDL

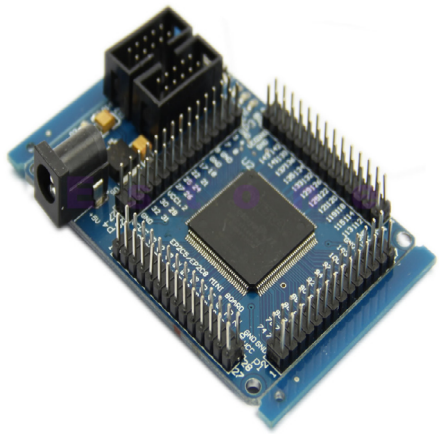
Najdete hned v další kapitole.

## 1.3 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 (v době psaní knihy).

https://datacipy.cz/c2ebay/
https://datacipy.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.

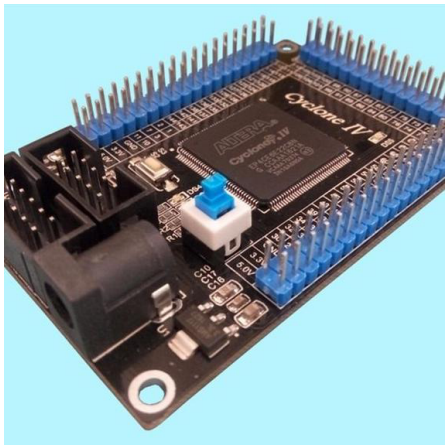


### Kity s Cyclone IV

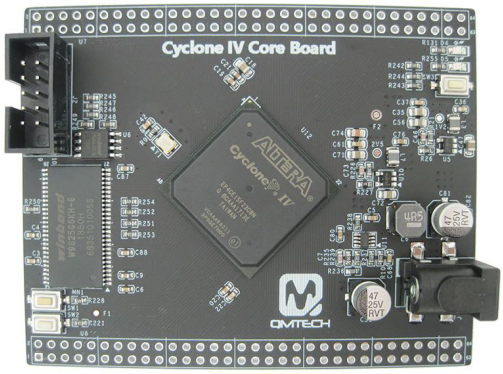
Výhoda kitů s čipy Cyclone IV je, že jsou jen o málo dražší, ale většinou podstatně vybavenější než ten nejmenší s Cyclone II. Často bývají integrovány různé periferie, jako LED nebo rozhraní pro VGA s konektorem, a někdy se objeví i integrovaná paměť SDRAM. Navíc podpora pro Cyclone IV je i v bezplatné verzi aktuální revize vývojového prostředí Quartus Prime Lite (v době psaní knihy šlo o verzi 19.1).

Můžete najít i jednoduché kity, podobné tomu předchozímu, za ceny okolo 700 Kč (v době psaní

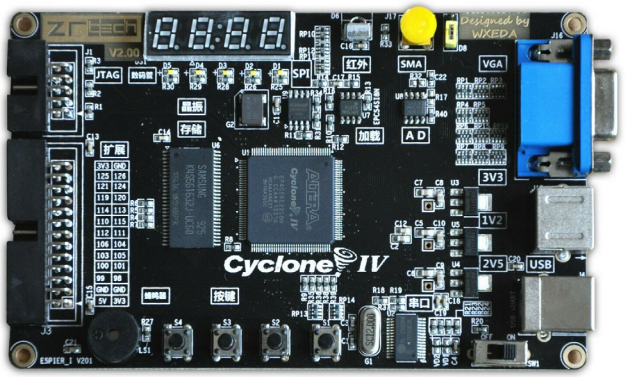
knihy). Hledejte klíčová slova „*FPGA Development Board EP4CE6E22C8N*“.



Za ceny do 1000 Kč lze koupit kit s větším čipem EP4CE15 a 32 MB SDRAM. Klíčová slova jsou, nepřekvapivě, „*EP4CE15 SDRAM*“

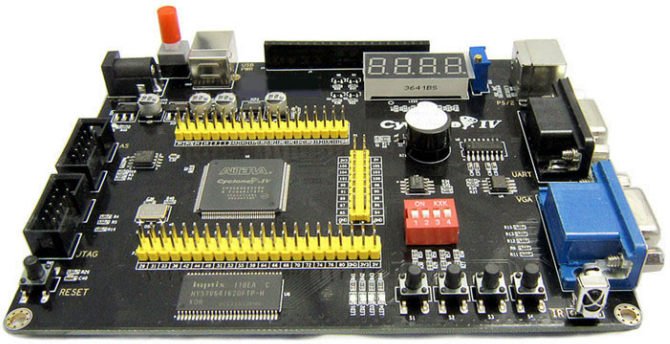



Lépe vybavené kity vás přijdou na částku lehce převyšující 1000 Kč, většinou tak do 1500 Kč. Není konkrétní výrobce ani převažující typ, spousta čínských výrobců si vyvíjí vlastní varianty, které se často liší od ostatních jen v detailech zapojení vývodů nebo rozmístění konektorů. Já například používám tento vývojový kit:




Většinou se jedná o různě upravené klony vývojových kitů Terasic DE (http://www.terasic.com.tw/), které se staly de facto standardem. I tyto kity můžete sehnat za ceny do 2000 Kč.

Já sám pro konstrukce, které vyžadují víc prostoru, používám „kit OMDAZZ" – nedělám si iluze o tom, že jde opět o nějaký *samo domo* klon čehosi, ale je dobře dostupný a dostatečně levný. Proto ho doporučuji i vám, pokud chcete zkusit nějakou náročnější konstrukci. Nevýhoda je, že není tak flexibilní jako kity „bez ničeho", výhoda naopak to, že obsahuje spoustu konektorů (sériový, PS/2, VGA), čidel (teplota, IR, tlačítka) i výstupů (LED, LED displej, konektor pro LCD displej, bzučák) ...




Podrobnější popis naleznete v příloze, když jej budete shánět, hledejte klíčová slova *omdazz* nebo *cyclone IV EP4CE6 board NIOSII FPGA*.

https://datacipy.cz/c4ebay
https://datacipy.cz/c4ali

*E-shop HW Kitchen by měl mít tento kit také v nabídce, více na jejich stránkách: https://hwkitchen.cz*

Pokud chcete větší a výkonnější čipy, můžete sáhnout např. po těchto deskách:

CYC1000 s Cyclone 10

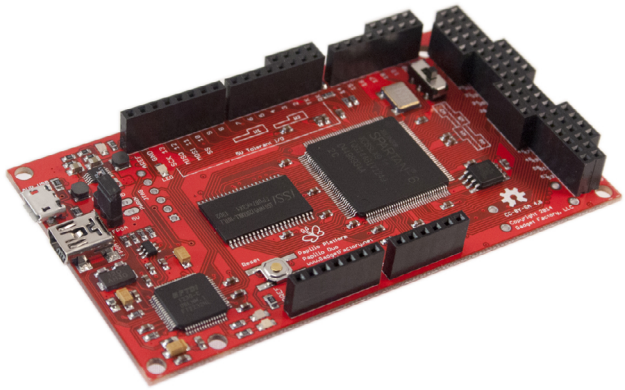
https://shop.trenz-electronic.de/en/TEI0003-02-CYC1000-with-Cyclone-10-FPGA-8-M-Byte-SDRAM

https://datacipy.cz/c10ebay

### Kity s obvody Xilinx (Spartan atd.)

Známý kit, který používá nejmenší Spartan 3, se jmenuje Papilio One. Používá obvody XC3S250 nebo XC3S500, které se od sebe liší především množstvím použitelných logických bloků a dedikované SRAM (24 kB, resp. 40 kB). Ve verzi Pro najdete desku se Spartanem 6 XC6SLX9 (ovšem za vyšší cenu, okolo 80 USD). Zajímavá deska od stejného výrobce je Papilio Duo. Představte si Arduino, jak jej znáte, s AVR ATMega32, a spolu s ním na jedné desce Spartan 6 (XC6SLX9), 512 kB (nebo 1 MB) SRAM, USB rozhraní a konektory, kompatibilní s Arduinem. Cena je okolo 100 USD, podle velikosti použité paměti.

http://store.gadgetfactory.net/fpga/



Za ceny okolo 30 USD můžete sehnat i poměrně slušně vybavené kity s obvody z řady Xilinx Spartan 6, především pak XC6SLX9.

https://datacipy.cz/x6ebay
https://datacipy.cz/x6ali

I kity se Spartanem 7 lze pořídit okolo tisícikoruny. Opět zajímavá volba může být Spartan Edge Accelerator – shield pro Arduino se Spartanem 7S15, obvodem ESP32 a standardním rozhraním pro kameru. Dodává jej Seeeduino za 36 USD (v době psaní knihy):

https://www.seeedstudio.com/Spartan-Edge-Accelerator-Board-p-4261.html

Do sta dolarů seženete i kity s čipy Artix 7.

https://store.digilentinc.com/cmod-a7-breadboardable-artix-7-fpga-module/

https://www.seeedstudio.com/Perf-V-Based-on-Xilinx-Artix-7-FPGA-RISC-V-opensource--p-4058.html

### Hotové konstrukce

Kromě popsaných kitů, které jsou určené primárně pro testování FPGA a vývoj s těmito čipy, existuje i kategorie zařízení takříkajíc „hackovatelných“. V těchto konstrukcích jsou použité přednastavené obvody FPGA, ale autoři více či méně úmyslně nechávají pokročilým uživatelům možnost nahrát vlastní konfiguraci a upravit si zařízení dle vlastní fantazie.

*Autor jedné takové konstrukce, Gameduina, v dokumentaci přímo psal: „Funguje jako grafická karta k Arduinu, ale když vás to znudí, můžete jej přeprogramovat a používat jako FPGA kit s Arduinem.“*

Takových zařízení je víc a některé z nich si ještě v knize představíme. Kromě zmíněného Gameduina nebo počítače V6Z80P (ani jedna konstrukce se už nevyrábí) to je například „reimplementace 8/16bitových počítačů“ MiST, Spectrum Next, nebo z poslední doby procesor MyMensch od společnosti Western Design Center.

*Zmíněná společnost, založená spolutvůrcem procesoru 6502 Billem Menschem, dodneška tento procesor vyrábí, a nabízí kromě fyzické podoby i jeho HDL implementaci. MyMensch je poměrně levný a jednoduchý kit s čipem Altera / Intel 10M08S z řady MAX10. Z výroby je v ní nahrána konfigurace, která odpovídá procesorům 65C02, 65C816 nebo 65C165, spolu s obvody VIA a ACIA. Díky vyvedenému konektoru JTAG můžete i tento kit překonfigurovat dle své libosti.*

## První pokus 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 první experiment 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 pro-

jektu existuje Project. Project v jednom místě schraňuje všechny soubory, potřebné k naprogramování FPGA. Jde především o popis funkce v některém z jazyků (VHDL, Verilog a další). K těmto 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 následujícímu:

- První krok je analýza a syntéza. V něm se ze zdrojových kódů vytvoří návrh pro konkrétní FPGA. Překladač zkontroluje syntax, 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 údaje z Pin Planneru. 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.

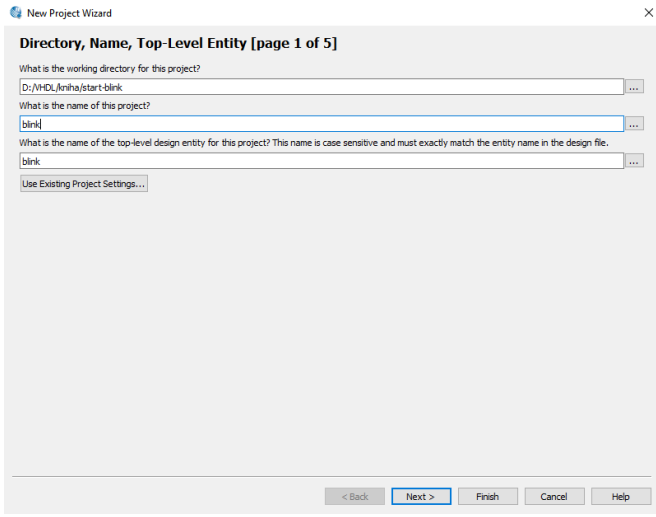
### Hello world, model FPGA

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

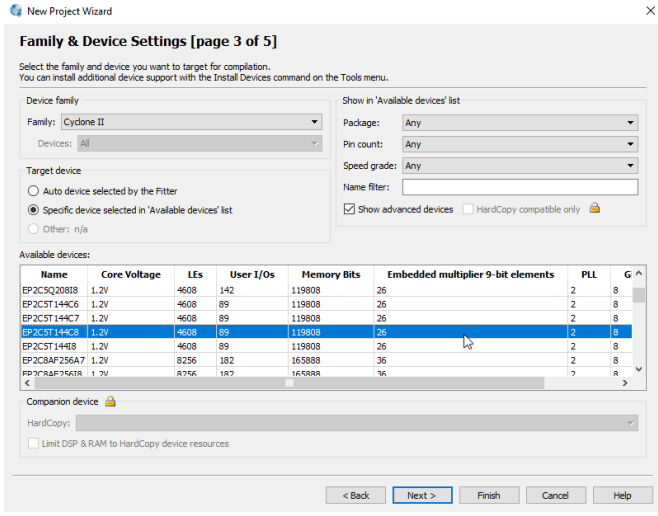
*Budu popisovat blikání LEDkou pro kit EP2C5, programátor USB Blaster a IDE Quartus.*

## 1. Vytvoření projektu

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




Projekt pojmenujeme „blink“ a pokračujeme (Next). Krok 2 jen přeskočíme (Next). Ve třetím kroku je potřeba vybrat použité FPGA. Zadejte rodinu „Cyclone 2“, čip je „EP2C5T144C8“.



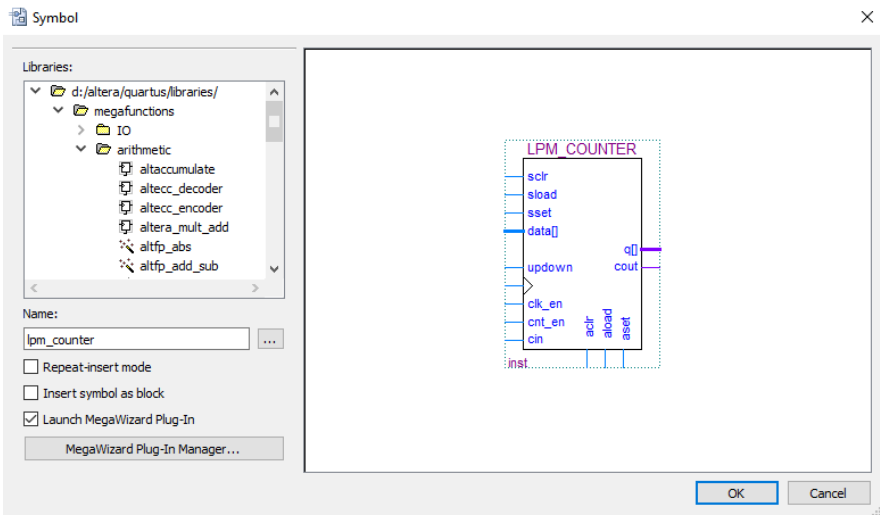

Next, next, finish…

## 2. Zapojení

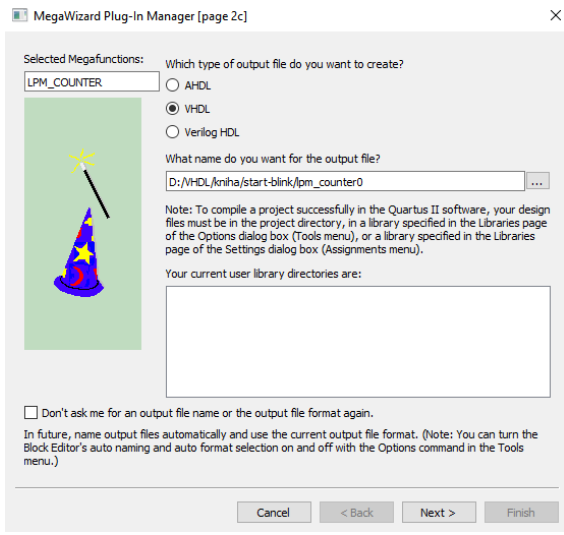
Blikání můžeme zařídit mnoha způsoby. Já vybral ten, kdy si na hodinový vstup (kam je připojený externí signál s kmitočtem 50 MHz) připojíme čítač, kterým podělíme frekvenci natolik, aby bylo blikání pozorovatelné pouhým okem. To znamená ideálně 24 bitů a víc. Kit má navíc 3 LED, takže nechám blikat všechny tři a zapojím je k čítači na výstupní bity 24, 25 a 26. Dělič nemusíme vytvářet z elementárních obvodů – Quartus obsahuje knihovnu (neskromně 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 nakreslíme si ho. Vyberte tedy File – New – Block Diagram/Schematic File. Otevře se známý „tečkovaný papír“, kam můžete umístit komponenty a propojovat je.

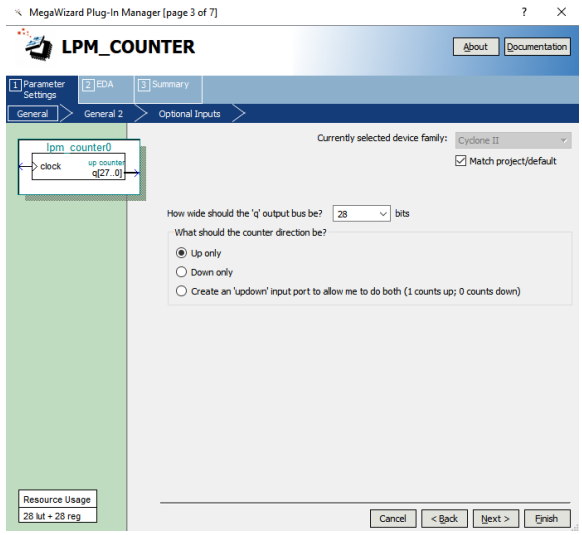
Nejprve tedy umístíme komponentu. Vyberte si z „Megafunctions“, složky „Arithmetic“ obvod, který se jmenuje „lpm_counter“.




Klikněte na OK, otevře se průvodce nastavením.

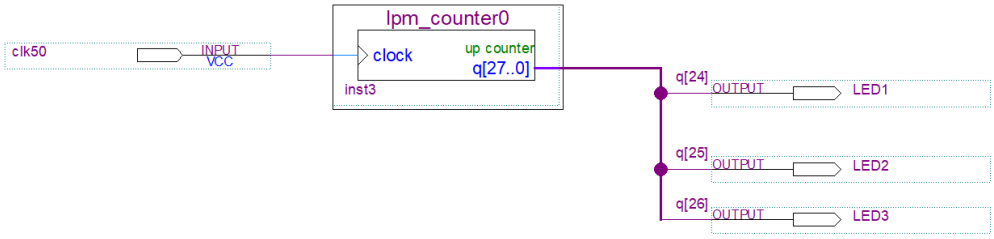



Jako jazyk zvolíme VHDL (teď je to jedno) a pojmenování necháme takové, jaké průvodce nabízí. Next. V dalším kroku zvolíme bitovou šířku. Já zvolil 28 bitů a čítání nahoru (mohu vytvořit i čítač dolů, popřípadě obousměrný). Další volby necháme tak, jak jsou nastavené, na konci klikneme na Finish.



Proč 28 bitů? Protože potřebuju, aby čítač zvládl napočítat takovou hodnotu, která bude blízká 50 milionům. 28bitové číslo může mít maximální hodnotu přes 268 milionů, což je dostatečný počet impulsů na to, aby byly vidět pouhým okem.

Teď nakreslíme zapojení. Na vstup připojíme vstupní pin, pojmenujme ho clk50. Na výstup připojíme sběrnici (Bus), která se bude jmenovat „q[27..0]“ – tedy má 28 linek. Využijeme z nich ale jen tři. Připravme si tři výstupní piny LED1-3 a připojme je ke sběrnici jako q[24], q[25] a q[26].

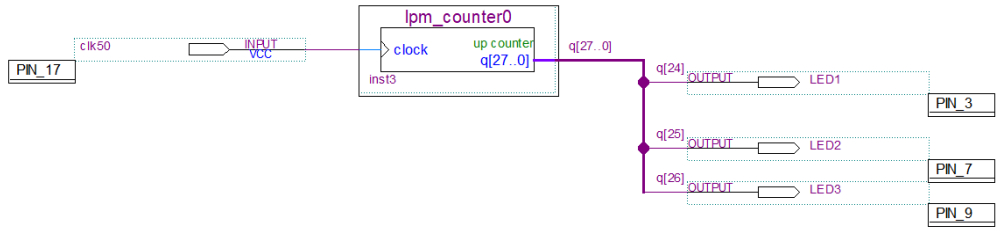



Soubor uložte jako „blink.bdf“, zkuste si zadat překlad (Processing – Start – Analysis & Synthesis, *Ctrl-K*), a pokud je vše OK, je načase připojit piny z nákresu k fyzickým. Otevřte si Pin Planner (Assignments – Pin Planner) a zadejte správné piny.

Jaké? Podle schématu kitu je hodinový vstup 50 MHz připojený na pin 17 a LED jsou připojené k pinům 3, 7 a 9. Stejně tedy přiřadíme i piny v planneru.

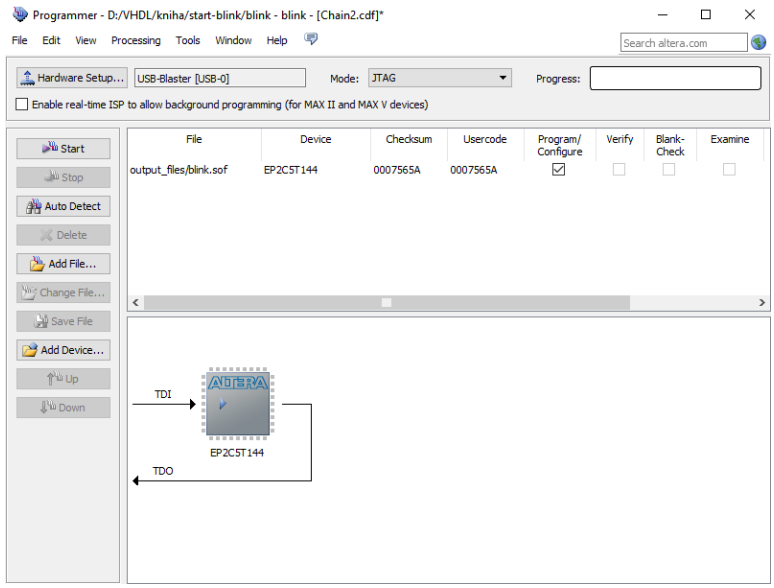
| Node Name | Direction | Location | I/O Bank | VREF Group | Fitter Location | I/O Standard | Reserved | Current Strength | Differential Pair | Weak Pull-Up Resistor |
|---|---|---|---|---|---|---|---|---|---|---|
| clk50 | Input | PIN_17 | 1 | B1_N0 | PIN_17 | 3.3-V LV...default) | | 24mA (default) | | |
| LED1 | Output | PIN_3 | 1 | B1_N0 | PIN_32 | 3.3-V LV...default) | | 24mA (default) | | |
| LED2 | Output | PIN_7 | 1 | B1_N0 | PIN_31 | 3.3-V LV...default) | | 24mA (default) | | |
| LED3 | Output | PIN_9 | 1 | B1_N0 | PIN_30 | 3.3-V LV...default) | | 24mA (default) | | |
| <<new node>> | | | | | | | | | | |

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



Teď tedy můžete 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 k externímu zdroji s napětím 5 V. Měla by svítit POWER LED. Otevřte programátor (Tools – Programmer), vyberte jako nástroj „USB Blaster“, mód „JTAG“ (pokud byste programovali natrvalo, viz výše, zde zadáte AS). Pokud se vám neukáže vpravo soubor, přidejte ho ručně (Add File, najdete jej v adresáři output_files pod názvem blink.sof). A pak už stačí jen kliknout na Start.




Během několika sekund se LED na kitu rozblikají. 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 první samostatné cvičení si můžete zkusit, jak do celého zapojení přidat tři invertory…

*Pro kit OMDAZZ použijte stejný postup, jen s několika změnami:*

- *Čip není Cyclone II, ale Cyclone IV, typ EP4CE6E22C8N*
- *Hodinový vstup CLK je na pinu 23*
- *LED jsou na pinech 85, 86 a 87*

# 2 Základy VHDL

# 2 Základy VHDL

## 2.1 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 schémata, 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.

*K tématu rozdílu mezi oběma jazyky se vrátím ještě na konci celé knihy.*

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 nejvíc 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 a Mac, 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 poštovným někde okolo pěti 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!

## 2.2 Než začneme…

Už v minulé kapitole jsem psal, co budeme potřebovat, ale pro jistotu připomenu:

- Quartus verze 13.0.1 Web Edition – v novějších není podpora pro čipy Cyclone II, které používáme. Aktuální (v době psaní knihy) verze (Quartus Prime Lite 19) umí Cyclone IV, takže kit OMDAZZ zvládne, ale Cyclone II už neumí.

- Nějaký kit – doporučuju levný základní kit s EP2C5, ale vyhoví jakýkoli, a jak říká legenda českého vaření: „Kdo nemá kit žádný, nepoužije žádný!“ (Ale připraví se o možnost zkusit si naučené v praxi.)

*Doporučený kit není dogma, stejně jako není dogma výrobce Altera / Intel. Pokud použijete kit s čipem Xilinx, použijte místo IDE Quartus odpovídající IDE od tohoto výrobce. Většinou se funkce jmenují podobně, ale nějaké drobné rozdíly jsou. Já budu v knize používat pro ilustraci „ekosystém Altera / Intel“. Kódy a příklady budou fungovat i s obvody Xilinx, maximálně s drobnými úpravami.*

## 2.3 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.

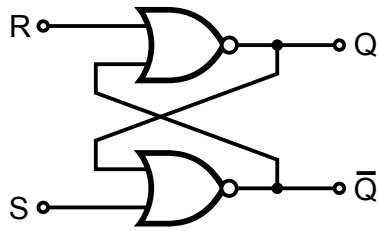
VHDL patří do rodiny jazyků HDL, což znamená „Hardware Description Language“. Návrháři už v názvu naznačují, že jde o jazyk popisný (description), nikoli programovací (programming) – *zdůrazňuju to proto, že programátoři mají tendenci každý „jazyk“ považovat za programovací, a pokud v něm nejde zapsat 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 číslicový obvod?

Máme několik možností, jak popsat číslicový obvod ve VHDL. Představme si takový klopný obvod R-S – pokud jste četli Hradla, volty, jednočipy, bude vám následující schéma povědomé:




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 dané 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. Musíme to teprve popsat. 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, nazvěme 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. (Skutečnost bude o něco málo složitější, protože ve VHDL nelze přímo propojovat vnitřní obvody, musíte si k tomu udělat *virtuální vodiče* – signály, ale k tomu se brzy dostaneme…)

### 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 lehce jinou techniku, ale princip je zhruba takovýto.*

### 3. Behaviorální popis

Tento styl popisu se asi nejvíc blíží klasickému programování a lidi, kteří mají zkušenost s programováním, budou mít tendenci vše řešit takto. Což je postup, před kterým varuju 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 na špatné adrese 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 to je nedefinovaný stav.* V procesu můžete použít právě podmínky, větvení, cykly a další programátorské vymoženosti, 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 popsat procesem.

### Chytáky pro programátory

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 i když to jsou nano- až pikosekundy, při vysokých rychlostech je s ním potřeba počítat. Pro tuto chvíli zpoždění zanedbejme.*

Představa, že *nejdřív něco změním, pak se něco provede, pak zase změním něco jiného* je ve světě logických 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. Odpovídá to realitě: v obvodech pracují všechny části současně a naráz, není tam nic, co by je postupně přepínalo mezi stavy.

*Dobrá analogie s reálným zapojením je „nepředstavujte si to jako příkazy, ale jako seznam propojovacích vodičů v dokumentaci". Tam je taky jedno, jestli propojení signálů D12 a SDA je zapsáno až PO propojení D11 a SCL, nebo PŘED ním, na funkci to vliv nemá.*

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 proměnné se přiřazuje pomocí := (ale velmi podobné přiřazení signálu se dělá pomocí <=) a celé to je silně typované. Jinak je VHDL tolerantní vůči velkým / malým písmenům, nijak neřeší mezery ani odsazení a je v tom velmi benevolentní.

Ještě takový terminologický detail: **Překlad,** tedy to, co z programování známe jako kompilaci kódu, tu tvoří několik kroků. Hlavní krok, podobný „kompilaci", je **syntéza,** kterou neprovádí kompilátor, ale **syntetizér.** On totiž nedělá to, že by kompiloval příkazy, on vytváří (syntetizuje) popsaný obvod podle popisu ve VHDL…

## 2.4 Hello world!

Lžu. Ještě ani zdaleka ne. Křivka učení je hodně povlovná a ještě musíme pár 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 základní koncepty.

Naše stavební bloky ve VHDL jsou entity. Entita odpovídá nějakému logickému celku; ve světě reálné elektroniky jí může odpovídat například integrovaný obvod. Entita je popsána jednak svým rozhraním navenek (viz dříve zmiňovaný **port**), jednak svou vnitřní architekturou. Ta může být popsána několika způsoby a v praxi se nejčastěji potkáte s jejich mixem.

Pojďme si nejprve nadefinovat entitu, která bude provozovat **neúplné jednobitové sčítání.** Jak to funguje?

Při sčítání dvou jednobitových hodnot je pravidlo prosté:

- 0+0 = 0
- 1+0 = 1 (a protože je sčítání komutativní, platí, že i 0+1=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 pravdivostní 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 XOR B*, Cout je *A AND B*. Můžeme zvolit strukturální 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 nazpaměť 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 sčítačka
```

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 |
| ‘-’ | 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_ulogic. Klasická „dvouhodnotová“ (ve skutečnosti jich má víc) logika je z ní odvozená a jmenuje se std_logic. Její použití *může* být nevýhodné, protože v některých situacích musí syntetizér používat *resolve* funkci, která jasně rozhodne, jestli je signál 0, nebo 1, což může návrh zesložitit a 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 právě z tohoto důvodu **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 podtržítko
- Začíná písmenem (ne číslicí ani podtržítkem)
- Nerozlišují se velká a malá písmena (ADDER je totéž co Adder)
- Nesmí končit podtržítkem
- Nesmí obsahovat dvě podtržítka za sebou (Takhle__Ne)

Deklarace je ukončena slovem „end“. Ve VHDL je mnoho ENDů (end if, end component atd.) Zde třeba „end entity“. Slovo „entity“ i její jméno jsou u „end“ nepovinné, ale mohou tam být. Radím zvyknout si, že vždy píšete, k čemu end patří. Já používám „zlatou střední variantu“, totiž „end entity“. Vím, co tento konkrétní END uzavírá, ale nemusím vypisovat název entity znovu a znovu.

Uvnitř deklarace je pouze část *port()*. Ještě se zde může vyskytnout část *generic(),* 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 tuto 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 víc, 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 nabyde hodnoty A xor B“ a „signál Cout nabyde 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.

## 2.5 LUT

Měl bych k vám být fér. A budu, takže vám už teď prozradím, že v obvodu FPGA není žádné supr čupr pole úplně volných hradel AND, NAND ani XOR a mezi nimi spousta konfigurovatelných kvantových vodičů. Místo toho tam jsou logické elementy, postavené kolem struktury, nazývané LUT.

LUT je zkratka z Look-Up Table, tedy česky něco jako „vyhledávácí tabulka". Ve skutečnosti to je obvod, který má několik vstupních a několik výstupních signálů a funguje skoro jako paměť EEPROM. Prostě pro všechny kombinace vstupních signálů dá na výstupu takový signál, který je v něm na dané adrese uložen. Syntezátor se postará o to, aby co nejvíc kombinačních obvodů dokázal vměstnat právě do takových LUT. Prostě vyzkouší všechny možné kombinace na vstupech, spočítá výsledek, a ten zanese do LUT. Ta se pak bude chovat přesně tak, jako syntetizovaný kombinovaný obvod.

LUT se spojují s dalšími jednoduchými obvody do takzvaných logických elementů (LE). U starších FPGA, třeba Cyclone II, je součástí takového elementu právě LUT, klopný obvod, pak nastavitelná logika, která se stará o obsluhu dalších systémů, a pár multiplexorů, kterými se nastavuje (napevno, v rámci syntézy) cesta signálu logickým elementem.

Je to možná „hack" a není to „ryzí hradlařina" – ovšem je to jednoduché a funkční!

## 2.6 Testování

Nastal čas otestovat sčítačku z předchozí kapitoly. Nejprve jsem si vytvořil v Quartu (pardon, ale klasické vzdělání mi brání psát v šestém pádu tvar „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 radím: Naučte se nazpaměť!) Entita se jmenuje „test“ a je prázdná – nenabízí 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“. Díky deklaraci překladač ví, že je někde nějaká externí entita „adder“, kterou použije jako komponentu pro sestavování aktuálního obvodu.

Pod komponentou (může jich zde být samozřejmě víc) 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 taková kouzla 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 sčítačky, 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 tolik 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.

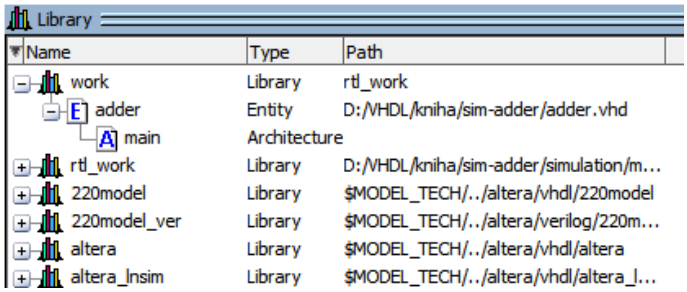
Pojmenování je „UUT", což je opět konvence pro testování: „Unit Under Test".

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).

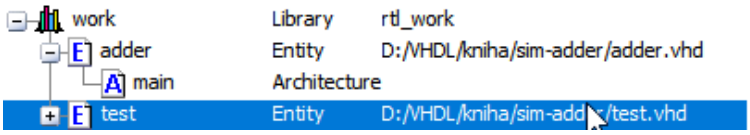
*Assignments – Settings – EDA Tool Settings a zde vybrat v řádku „Simulation" možnost „ModelSim – Altera". Pokud bude později Quartus protestovat, že není zadána cesta, zvolte „13.0sp1\modelsim_ase\win32aloem".*

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

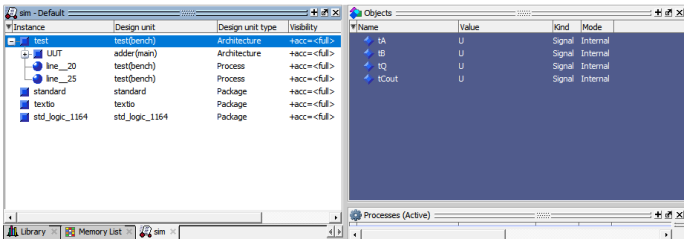



Teď musíme přeložit samotné testovací zapojení.

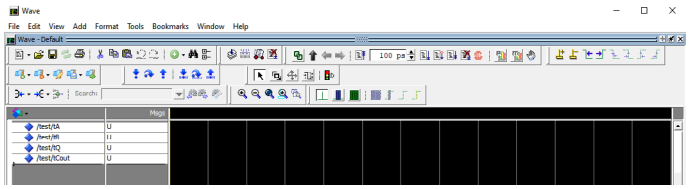
V menu Compile vyberte Compile, najděte soubor testbench.vhd (bude o dvě úrovně adresáře výš – dialog se otevře v podadresáři „simulation/modelsim“, což je místo, kam si příště své testovací soubory ukládejte) a dvojitým poklepáním soubor přeložte. Pokud je bez chyb, přidá se do seznamu v knihovně „work“:



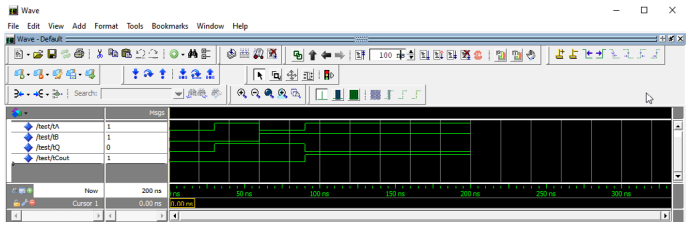

Klikněte pravým tlačítkem myši na „test“ a vyberte „simulate“. Okna se budou chvilku přeskupovat, a nakonec uvidíte něco, co bude velmi podobné následujícímu obrázku:




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. Vyberte je myší, klikněte pravým tlačítkem a zadejte „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 přepněte Wave ze zobrazení v panelu na samostatné okno. V něm proběhne vlastní simulace.



Nahoře uprostřed vidíte hodnotu „100 ps“ – tedy sto pikosekund. To je čas, který 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, a pak nechte proběhnout ještě dalších 100 ns.



Vidíte, že vše funguje tak, jak má. Signály A a B se mění tak, jak jsme zapsali, a sčítačka 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 víc 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“ nasimulovat v duchu hesla „takhle nějak by to vypadalo, kdyby se to spustilo“?)*

## GHDL

Existují i nástroje, které dokáží emulovat chod testovacího skriptu (testbench) přímo z příkazové řádky, bez nutnosti spouštět velké IDE. Jedním z nich je třeba GHDL:

https://github.com/ghdl/ghdl

GHDL si můžete přeložit, popřípadě můžete stáhnout už přeložený nástroj. Doporučím pro

začátek stáhnout pro Windows verzi „mingw32-mcode“, nikoli tu „-llvm“, protože „-mcode“ funguje i bez dalších nutných nástrojů.

Nejprve si připravme testovací zapojení – samotný obvod a testbench. Použijeme už definovanou komponentu adder (adder.vhd) a testbench (adder_tb.vhd).

Nejprve je potřeba oba elementy přeložit:

```
> ghdl.exe -a adder.vhd
> ghdl.exe -a adder_tb.vhd
```

*Pozor! GHDL (aktuální verze v době psaní knihy je 0.37) má problém se znaky s diakritikou, a to i v komentářích, bohužel...*

Po překladu obou prvků (přepínač -a) je načase zkusit vygenerovat report. Použijeme přepínač -r a jako parametr dáme název entity, v našem případě „test“:

```
> ghdl.exe -r test
```

Pokud je vše v pořádku, test proběhne a nic se nestane. Což je sice dobrá zpráva (nemáme syntaktické chyby a asi se něco i stalo), ale co když se nestalo nic? Je načase přidat nějaká hlášení a kontroly.

Přepíšeme testbench tak, aby používal „proces“. Nebojte se, později se k tomu vrátíme a vysvětlíme si princip procesu, pro tuto chvíli můžeme předpokládat, že „proces“ je kód který se provádí sekvenčně, což je to, co potřebujeme.

Celý začátek „adder_tb.vhd“ může zůstat stejný až po slovo „begin“. Za tímto slovem začínala část, která přiřazovala hodnoty signálům tA a tB a měnila je v čase. Tuto část přepíšeme takto:

```
testing: process
    begin
        tA <= '0'; tB <= '0'; wait for 10 ns;
        tA <= '0'; tB <= '1'; wait for 10 ns;
        tA <= '1'; tB <= '0'; wait for 10 ns;
        tA <= '1'; tB <= '1'; wait for 10 ns;
        report "Test OK";
        wait;
    end process;
```

Znovu přeložíme a spustíme:

```
> ghdl.exe -a adder_tb.vhd
> ghdl.exe -r test
```

Už to něco vypsalo – že vše bylo OK. Ale bylo to opravdu tak?

Přidáme výpis hodnot. Použijeme k tomu opět příkaz „report“, ale ten vyžaduje jeden parametr typu řetězec. Když napíšeme „report tQ;“, při překladu vyskočí chyba. Trik je *přetypování* – k němu se taky dostaneme později. Pro tuto chvíli stačí vědět, že postup je:

```
report std_logic'image(tQ);
```

A když už jsme v tom, tak si vypišme rovnou vše (řetězce se spojují operátorem „&“:

```
report std_logic'image(tA) & " + " & std_logic'image(tB) & " = " & std_logic'image(tCout) &
std_logic'image(tQ);
```

Tento řádek si můžeme zkopírovat za každé nastavení hodnot, což je ale nepraktické. Místo toho si nadefinujeme proceduru, takže výsledek bude vypadat takto:

```
testing: process

  procedure vypis is
  begin
      report std_logic'image(tA) & " + " & std_logic'image(tB) & " = " & std_
logic'image(tCout) & std_logic'image(tQ);
  end procedure;

  begin
    tA <= '0'; tB <= '0'; wait for 10 ns;
    vypis;
    tA <= '0'; tB <= '1';  wait for 10 ns;
    vypis;
    tA <= '1'; tB <= '0';  wait for 10 ns;
    vypis;
    tA <= '1'; tB <= '1';  wait for 10 ns;
    vypis;
    report "Test OK";
    wait;
end process;
```

Opět přeložíme a spustíme:

```
> ghdl.exe -a adder_tb.vhd
> ghdl.exe -r test
adder_tb.vhd:25:9:@10ns:(report note): '0' + '0' = '0''0'
adder_tb.vhd:25:9:@20ns:(report note): '0' + '1' = '0''1'
adder_tb.vhd:25:9:@30ns:(report note): '1' + '0' = '0''1'
adder_tb.vhd:25:9:@40ns:(report note): '1' + '1' = '1''0'
adder_tb.vhd:37:9:@40ns:(report note): Test OK
```

To už vypadá smysluplně a prostým okem je vidět, že hodnoty jsou správné. Sčítačka funguje!

*Mimochodem, v příkladech ke knize na webu najdete i soubor test, popř. test.bat – v něm jsou právě výše uvedené příkazy, takže stačí napsat jen „test adder", a správně se provedou všechny nezbytné operace, tj. překlad komponenty, překlad testbenchu a spuštění.*

VHDL jde samozřejmě ještě o kousek dál a místo ručního testování, jestli je vše tak, jak má být, můžeme použít příkaz *assert*. Ten zkontroluje zadané hodnoty, a pokud neodpovídají požadovaným, vypíše hlášení.

```
tA <= '0'; tB <= '0'; wait for 10 ns;
assert tQ = '0' and tCout = '0' report "0+0 failed" severity failure;
```

Nastaví se hodnoty tA a tB, simulátor počká 10 nanosekund a pak *se tvrdí,* že tQ = '0' a tCout = '0'. Pokud ano, nic se neděje, jede se dál. Pokud ne, tak se vypíše hlášení o tom, že selhalo sčítání 0+0. Za reportem jsou ještě uvedena dvě slova „severity failure", která říkají, že toto hlášení má nejvyšší závažnost a dál se v simulaci nemá pokračovat. Další úrovně jsou „error", „warning" a „note", s klesající úrovní závažnosti.

Díky konstrukci assert a nástrojům jako GHDL můžete psát automatizované testy, v nichž je rovnou vidět, jestli vše funguje, nebo jestli se vyskytl nějaký problém.

*Tip: pokud narazíte na nečekaný problém při testu, například na oznámení, že nelze volat nějakou funkci, která je podle všeho v pořádku, může být důvod v tom, že GHDL používá standard VHDL-93 a některé konstrukce jsou k dispozici až od verze VHDL 2008. V takovém případě použijte přepínač --std=08. Při použití nestandardizovaných knihoven můžete narazit na podobný problém, pomůže přepínač --fsynopsys.*

```
> ghdl.exe -a --std=08 --fsynopsys adder_tb.vhd
```

Návod ke GHDL naleznete na https://ghdl.readthedocs.io/

### GTKWave

Další nástroj, který může být užitečný při testování, je GTKWave.

https://sourceforge.net/projects/gtkwave/

Jak už název napovídá, slouží k vizualizaci průběhů, podobně jako jsme si ukazovali výše. Jde o open-source nástroj, volně ke stažení a použití.

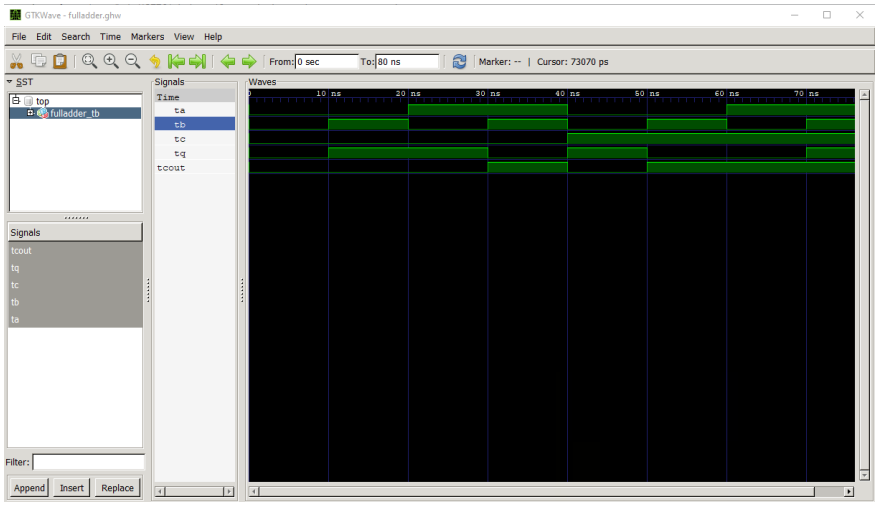
Vstupní data pro GTKWave získáte jednoduše z GHDL – stačí připojit k příkazu pro testování (ghdl -r) parametr, který určuje, kam má GHDL uložit průběhy:

```
> ghdl.exe -r test --wave=adder.ghw
```

nebo

```
> ghdl.exe -r test --vcd=adder.vcd
```

Výsledný soubor .ghw (GHdl Wave) nebo .vcd můžete poté otevřít v GTKWave, vybrat si signály, které chcete sledovat, a podívat se na jejich průběh během emulace.



### Konvence pro zdrojové kódy

V kódech ke knize dodržuju jednotnou jmennou konvenci:

- Komponenta „mux“ je v souboru „mux.vhd“
- Testbench k této komponentě je v souboru „mux_tb.vhd“
- Testovací komponenta samotná se jmenuje „mux_tb“

Díky tomu mohu definovat testovací příkaz „test“, který vezme název komponenty („test mux“), správně přeloží mux.vhd i mux_tb.vhd a vyhodnotí běh komponenty mux_tb.

## 2.7 Komponenty a signály

Už jsme na obojí narazili. Pojďme 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 obsahuje několik multiplexorů. Multiplexory 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.

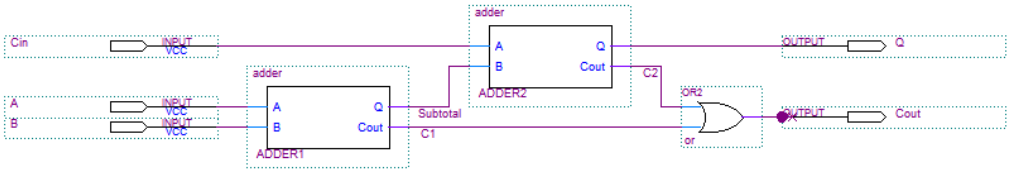
Co ale v situaci, kdy chceme připojit (například) vstup jednoho hradla na výstup druhého? Narazíme na to, že pro podobné propojení „nemáme jméno do port mapy". VHDL proto zavádí koncept **signálu.** Signál je „interní vodič" – můžete si ho představit jako fyzický vodič, kterým jsou propojeny jednotlivé 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 „vstupně – výstupních signálech" (až doteď 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 záludnost, 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, pojďme 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 sčítačku z hradel, nebo můžeme zvolit ten přístup, kdy pomocí naší neúplné sčítačky 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).

| A | B | Cin | Q | Cout |
|---|---|---|---|---|
| **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 sčítačky a vstupem B druhé (Subtotal), a pak mezi jednotlivými přenosy a hradlem OR (C1 a C2). Pojďme si tedy napsat plnou sčítačku. 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 sčítačce, jen je přidaný port Cin. Druhá část bude architektura. Začneme hlavičkou:

```
architecture main of fulladder is
```

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 sčítačky (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 tolik 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) sčítačka. 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 sčítačky 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 se rozloučit:

```
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 jako 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 rovnítkem, nikoli zdvojeným (to je 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 později, 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 sčítačky 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 (popřípadě XNOR).

### „Help I Accidentally Build A Latch“

Jestli vám až do této chvíle připadal latch (česky *závora*, bližší popis najdete třeba v knize Hradla, volty, jednočipy) jako úplně normální součástka, jakých jsou plné katalogy (SN7475 například), tak po téhle podkapitole se váš pohled na něj změní.

Latch je obvykle 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 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 sčítačku 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 větvi 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.

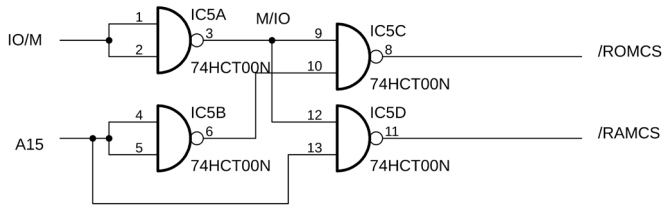
Jenže elektronický obvod nemá nic jako „neměň výstup“, a pokud chci, aby se výstup nezměnil, tak si musím jeho hodnotu někde zapamatovat. Kde? No, zkuste hádat! A aniž byste to chtěli, tak vznikne latch, většinou zcela nadbytečný, a zabírá místo v návrhu. „Omylem vytvořený latch“ je něco jako memory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

*Na tento internetový mem odkazuje i název podkapitoly. Vyhledávač Google totiž při vyhledání písmen „help i ac“ navrhoval vyhledání výrazu „help i accidentally build a shelf“, tedy „pomoc, náhodou jsem postavil regál“. Představuju si situaci, která vedla k tomu, že někdo usedl ke Googlu a zadal přesně tento dotaz...*

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!**

## Prakticky: Jednoduchý kombinační obvod

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 a přímočaře:

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

architecture main of alphaDecoder 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 behavioral of alphaDecoder is
begin

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

end architecture;
```

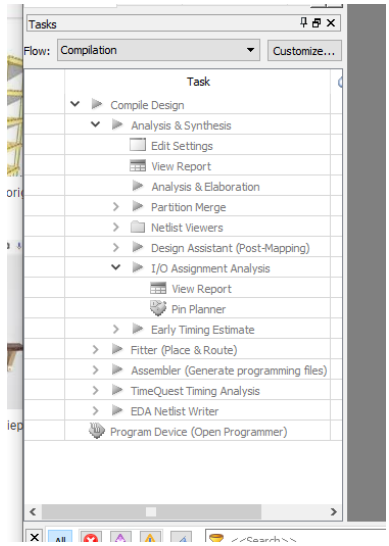
Vidíme, že /ROMCS je 0, pokud je A15 = 0 a IOM = 0, jinak je 1, obdobně pro /RAMCS.

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 nalezl, 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...

Ale samozřejmě si připravím i testbench:

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

entity testbench is
end;

architecture bench of testbench is

component alphaDecoder is
    port (
        IOM, A15: in std_logic;
        nRAMCS, nROMCS: out std_logic
    );
end component;
```

```
  signal io, addr, ram, rom: STD_LOGIC;

begin


testing: process

procedure vypis is
 begin
   report "IO/M:" & std_logic'image(io) &
          ", A15:" & std_logic'image(addr) &
          " => nRAMCS=" & std_logic'image(ram) &
          " => nROMCS=" & std_logic'image(rom);
end procedure;

  begin
      io <= '0'; addr <= '0'; wait for 10 ns;
      vypis;
      io <= '0'; addr <= '1'; wait for 10 ns;
      vypis;
      io <= '1'; addr <= '0'; wait for 10 ns;
      vypis;
      io <= '1'; addr <= '1'; wait for 10 ns;
      vypis;
      wait;
  end process;

UUT: alphaDecoder port map (io, addr, ram, rom);

end bench;
```

## Cvičení

Zkuste si navrhnout dekodér pro sedmisegmentovky. Pomocí znalostí, které už máte, to půjde. Ale bude to velmi neefektivní. Hodilo by se mít možnost pracovat s více bity najednou, že? To si ukážeme hned v další kapitole.

## 2.8 Bit sem, bit tam...

I počítače jsou alespoň osmibitové. Buďme i my vícebitoví!

Až dosud jsme si ukazovali všechno jednobitové: Jednobitová sčítačka s jednobitovými daty, jednobitové 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 číselné hodnoty.

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 bity
   -- a hodnoty pro ostatní (nevyjmenované)

DBUS <= (others=>'0');
   -- všechny bity 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 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), například *DBUS(0).*

## Čtyřbitová sčítačka

Pokračujme v našem příkladu a sestavme si ze čtyř jednobitových sčítaček jednu čtyřbitovou. Její zapojení nepřekvapí: Dvě vstupní hodnoty A a B budou tentokrát čtyřbitové vektory, totéž výstup Q. Cin je připojen na vstup Cin sčítačky nejnižšího řádu, Cout na výstup Cout sčítačky 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 adder4B 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 main of adder4B 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 sčítačky obsahuje komponentu fulladder, definovanou v minulé kapitole, a čtyři interní signály C0 až C3, pomocí kterých budeme propojovat výstup Cout jedné sčítačky se vstupem Cin druhé. V těle jsou pak vytvořeny čtyři instance jednobitové sčítačky a 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 sčítačka funguje, teoreticky, bez problémů.

*V reálném světě bych se takto zapojené sčítačce raději vyhnul, protože má jednu výraznou nectnost, 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 podrobněji.*

Samozřejmě otestujeme... Připravíme si testbench, vyjdeme z toho, který už máme, pouze signály tA, tB a tQ změníme na vektory.

Testování pomocí assert je bez problémů:

```
tA <= "0000"; tB <= "0000"; tC <= '0';
wait for 10 ns;
assert tCout = '0' and tQ = "0000"
       report "0+0+0 failed" severity failure;
```

Problém nastane při výpisu – report odmítá vypsat vektor, a jeho konverze na řetězec naráží na

problémy. Proto si ukážeme, bez dalšího vysvětlování, jeden trik – funkci *toString*, která zkonvertuje vektor na řetězec:

```
function toString ( a: std_logic_vector) return string is
  variable b : string (1 to a'length) := (others => NUL);
  variable stri : integer := 1;
  begin
    for i in a'range loop
      b(stri) := std_logic'image(a((i)))(2);
    stri := stri+1;
    end loop;
  return b;
end function;
```

S touto funkcí můžeme snadno upravit proceduru *vypis*:

```
procedure vypis is
  begin
    report toString(tA) & " + " &
           toString(tB) & " + " &
           std_logic'image(tC) &
           " = " &
           std_logic'image(tCout) &
           toString(tQ);
end procedure;
```

Vraťme se zpátky k naší sčítačce a pojďme si ji 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), C(0), Q(0), C(1));
A1: fulladder port map (A(1), B(1), C(1), Q(1), C(2));
```

```
A2: fulladder port map (A(2), B(2), C(2), Q(2), C(3));
A3: fulladder port map (A(3), B(3), C(3), 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 zápise jednotlivých instancí určitou logickou strukturu. A pokud se těšíte, že budete moci sčítačky 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 bity ve vektoru. Pro rozsah *0 to 3* vzniknou čtyři sčítačky, jejichž porty budou nastaveny naprosto 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 určená *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), C(N), 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 deklarací 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 nelíbí 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 šířkou v bitech, a k nim máte definované základní matematické operace. Takto bychom čtyřbitovou sčítačku nadefinovali tak, že nepoužívá vektory, ale „unsigned (3 downto 0)", a samotné sčítání by bylo „Q <= A + B + Cin;" – takhle prosté, protože máme „plus" definované. Syntetizér si s tím už nějak poradí, a pokud má daný obvod například integrované hardwarové sčítačky, 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 zaplácaný 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 navrhnul 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  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 uveďte:

```
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 přetypovávat… 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 sčítačky), tam patří numeric_std (ne arith). Tam, kde vícebitový signál nemá rozměr čísla, tam použijte std_logic_vector (někdy se též setkáte se zkráceným označením SLV). Například osmibitový vstup multiplexoru nebo sběrnice pro přerušení IRQ0 – IRQ7 jsou typické případy, kdy neříkáme „přišlo přerušení s hodnotou 16“, ale „přišlo přerušení IRQ4“ – tedy SLV.

Pokud je potřeba převést jeden typ na druhý, tak se nevyhnete přetypování. Ale snažte se nepoužívat std_logic_arith, std_logic_unsigned a std_logic_signed. Tyto knihovny jsou zavržené (deprecated), navíc jsou méně flexibilní než numeric_std.

## 2.9 Typy, operátory a atributy

Nadešel čas na trochu ryzí teorie… 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*. Pojďme 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 |
|---|---|---|
| **std_ulogic** (sul) | U, X, 0, 1, Z, W, L, H, - | std_logic_1164 |
| **std_ulogic_vector** (sulv) | Vektor (pole) hodnot typu std_ulogic | std_logic_1164 |
| **std_logic** (sl) | jako std_ulogic, resolved (tj. rozhodované hodnoty) | std_logic_1164 |
| **std_logic_vector** (slv) | Vektor hodnot typu std_logic | std_logic_1164 |
| **unsigned** (uv) | SLV N bitů, rozsah 0 .. $2^{N-1}$ | numeric_std |

| | | |
|---|---|---|
| **signed** (sv) | SLV N bitů, rozsah $-2^{N-1}$ .. $2^{N-1}-1$ | numeric_std |
| **boolean** | výčet (true, false) | standard |
| **character** | znak ASCII | standard |
| **string** | pole znaků | standard |
| **integer** | 32bitové signed číslo ($-2^{31}$ - 1 .. $2^{31}$ - 1) | standard |
| **real** | -1.0E38 .. 1.0E38 | standard |
| **time** | 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).

Některé typy jsou tzv. „resolved“, jiné „unresolved“, česky bychom řekli „rozhodované“ a „nerozhodované“. Signál rozhodovaného typu typ (resolved) může mít přiřazen několik „budicích výrazů“ najednou, a o jeho hodnotě je potřeba rozhodnout. Nerozhodovaný (unresolved) signál může být buzen právě jedním výstupem.

Resolved typy (std_logic např.) tak umožňují například konstruovat zapojení typu „montážní OR“ apod.

Představme si komponentu – jednoduché „hradlo“ s dvěma vstupy a jedním výstupem, které je udělané tak, že „prostě spojí dva vstupy dohromady“:

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;
ENTITY resolved IS
    PORT (
        A, B : IN std_ulogic;
        Q : OUT std_logic
    );
END ENTITY;
```

```
ARCHITECTURE main OF resolved IS
BEGIN
    Q <= A;
    Q <= B;
END ARCHITECTURE;
```

Všimněte si, že výstup Q je typu std_logic, tedy „resolved“, proto překladač nevyhodí chybu v architektuře, kde tomuto signálu přiřazujeme signál A i signál B. Jen tak.

Kdyby Q bylo typu „std_ulogic“, nastane chyba. Ale s resolved typem „std_logic“ je taková konstrukce dovolená – ovšem *někde* musí existovat mechanismus, který „rozhodne“, jak to tedy bude. V tomto případě existuje, je součástí standardní knihovny, a říká, jak se v takovém případě má s podobnou konstrukcí naložit.

Zkusme si postavit testbench *resolved_tb*:

```
ENTITY resolved_tb IS
END;

ARCHITECTURE bench OF resolved_tb IS

    SIGNAL tA, tB : STD_ULOGIC;
    SIGNAL tQ : std_logic;

BEGIN
    testing : PROCESS

        PROCEDURE vypis IS
        BEGIN
            REPORT std_ulogic'image(tA) & " + " & std_ulogic'image(tB) &
                " = " & std_ulogic'image(tQ);
        END PROCEDURE;
    BEGIN
        tA <= '0'; tB <= '0'; WAIT FOR 10 ns; vypis;
        tA <= '0'; tB <= '1'; WAIT FOR 10 ns; vypis;
        tA <= '1'; tB <= '0'; WAIT FOR 10 ns; vypis;
        tA <= '1'; tB <= '1'; WAIT FOR 10 ns; vypis;

        tA <= 'Z'; tB <= '0'; WAIT FOR 10 ns; vypis;
        tA <= 'Z'; tB <= '1'; WAIT FOR 10 ns; vypis;
```

```
        REPORT "Test OK";
        WAIT;
    END PROCESS;

    UUT : ENTITY work.resolved PORT MAP(tA, tB, tQ);

END bench;
```

Po spuštění simulace uvidíme, jak si s tímto „spojením nakrátko“ simulátor poradil:

```
resolved_tb.vhd:17:13:@10ns:(report note): '0' + '0' = '0'
resolved_tb.vhd:17:13:@20ns:(report note): '0' + '1' = 'X'
resolved_tb.vhd:17:13:@30ns:(report note): '1' + '0' = 'X'
resolved_tb.vhd:17:13:@40ns:(report note): '1' + '1' = '1'
resolved_tb.vhd:17:13:@50ns:(report note): 'Z' + '0' = '0'
resolved_tb.vhd:17:13:@60ns:(report note): 'Z' + '1' = '1'
resolved_tb.vhd:48:9:@60ns:(report note): Test OK
```

Vidíme, že dvě nuly nebo dvě jedničky jsou bez problémů vyhodnocené. Stejně tak logický signál a vysoká impedance Z. Kombinace nuly a jedničky vede k nerozhodnutému stavu „X“.

Nulu a jedničku si můžeme představit jako „silné“ hodnoty, připojené přes minimální odpor k napájecímu napětí nebo zemi. Co když ale použijeme hodnoty slabé, H a L (odpovídající zapojení přes nějaký větší rezistor)?

```
resolved_tb.vhd:17:13:@50ns:(report note): 'H' + '0' = '0'
resolved_tb.vhd:17:13:@60ns:(report note): 'H' + '1' = '1'
resolved_tb.vhd:17:13:@70ns:(report note): 'H' + 'L' = 'W'
resolved_tb.vhd:17:13:@80ns:(report note): 'H' + 'H' = 'H'
```

„Slabé H“ (představme si takový signál jako pull-up rezistor) s nulou dá ve výsledku nulu, s jedničkou jedničku. H s L dohromady dá „slabý neutrální signál“ W…

Resolved signály mohou svádět k vytváření podobných „divokých“ zapojení, ale radím: **vyhněte se jim, pokud to není opravdu nezbytné.** Nutnost rozhodování může vést k pomalejšímu a neefektivnímu překladu.

Rozhodování už jde trochu nad rámec této knihy, zájemce odkazuju na shrnutí VHDL v příloze a na další literaturu.

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. Vyhněte se prosím podobným nejednoznačnostem.

### 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 víc položek oddělených čárkou, pak vznikne vícerozměrné pole. Jako specifikace může být použit i výčet…

### Záznamy

Ano, Pascal opět vystrkuje růžky. Záznam (record) je složený, tedy 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** | |
|---|---|
| **Exponent, absolutní hodnota** | **, abs |
| **Logické funkce** | and, or, nand, nor, xor, xnor, not |
| **Multiplikativní funkce** | *, /, mod, rem |
| **Aditivní funkce, negace** | +, – |
| **Spojování, posuny, rotace** | &, sll, srl, sla, sra, rol, ror |
| **Relační operátory** | =, /=, <, <=, >, >= |

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 přemýšlíte, 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 <= A (15 downto 8)*. Rozsah vybereme jeho uvedením v závorce za signálem.

Že se vracím ještě k té sčítačce:

```
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 k dispozici). Dělení, modulo a operátor zbytku (rem) bývají hůř syntetizovatelné…

**Posuny** jistě znáte, ale připomenu: Logické (sll, srl) zaplňují volná místa nulami, aritmetický posun vpravo (sra) opakuje nejvyšší 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 rozsahu stoupající - to (true), nebo klesající - downto (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'*.

Některé atributy slouží pro konverzi hodnot:

```
INTEGER'IMAGE(x) – převede hodnotu X typu INTEGER na řetězec
INTEGER'VALUE(x) – převede řetězec X na hodnotu typu INTEGER
```

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

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

## 2.10 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 znovu upozornit na jednu důležitou věc.

Představme si, že máme nějaký obvod, třeba AND-OR-INVERT – čtyřvstupový obvod, který provádí operaci *Q = 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. Teď už bychom měli vědět, jak ho popsat. Pojďme na to. 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 rsko 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 vznikají signály Q a nQ, ale „Co se stane, když na vstup R přijde 1? A když přijde na vstup S?" U R-S obvodu si dokážeme představit jeho obvodové zapojení, ale jsou situace, kdy dokážeme říct, co má obvod dělat, ale nechce se nám ho rozkládat na komponenty a popisy signálů. V dalších kapitolách takových situací zažijeme ještě spousty.

Klíčovým pojmem behaviorálního popisu je **proces.** S procesem už jsme se krátce setkali při testování, když jsme si ukazovali nástroj GHDL, a sliboval jsem, že se k němu vrátíme.

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. Proto se někdy nazývá jako *sekvenční kód* – v protikladu ke kódu *konkurenčnímu*, což v kontextu VHDL neznamená nic nekalého ani nepřátelského, je to prosté označení pro *souběžné zpracování* (jak jsem už psal: jako kdyby vše probíhalo naráz, není žádné „předtím" a „potom"). 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 *bohužel!* Proces se sice provede sekvenčně, ale pamatujte si, že hodnota se signálům nastaví „až pak, někdy na konci". Syntetizátoru je úplně jedno, jak šibujete 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 :=
- 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), deklarujeme 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 <=, nebo proměnné pomocí :=

### 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 elsif)
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 splněná 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), buď v podmíněné větvi nebo třeba ve tvaru *exit when (podmínka)*. Obdobou příkazu „continue“ je příkaz *next* – tedy další iterace. Opět možno zapsat jako *next when (…)*

### Příklad procesu

Vraťme se ještě k našemu klopnému obvodu R-S. Jak ho popsat behaviorálně? 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 proměnnou *state* – to je pro mne „interní stav klopného obvodu“, tedy proměnná, 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 podmínkách 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 behaviorální, ale v níž jsem místo proměnné použil signál, abych názorně předvedl rozdíl mezi proměnnou a signálem.

```
architecture behavioralS of rsko 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. Tipnete si, co se stane?

Testbench je zde:

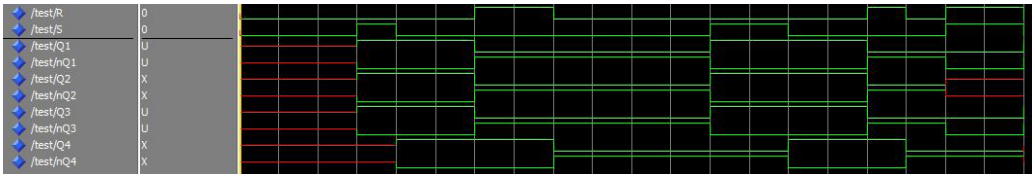
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

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 chvilku 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 hloupost, navíc máte kód zapráskaný duplicitními zápisy 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 samostatné pojmenované 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 děly správně, ale pozdě! Proč?

Když se nad tím zamyslíte: problém se skrývá v tom, co jsme si tu už řekli jen tak mimochodem 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 sčítačka

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

Vzpomínáte na generickou sčítačku, 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!*

## 2.11 Hodinové signály a čas

Tušíte správně, 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 mém 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 přičtu 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 použ*ít 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 featura 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 teď signál CLK = '1'… pak ta událost logicky musela být vzestupná hrana.

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

*Podrobněji si o možnostech programování reálného kitu řekneme víc v kapitole Hardware, pokud chcete programovat hned teď, můžete si ji nalistovat...*

### Alternativní blikání

Napadla mě ještě alternativa, při které si architekturu rozdělím na dvě části, na asymetrickou děličku 1:50000000 (má nestejné 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 podmínku 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." Jinak – náhodou postavený regál!*

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 čísly, které v desítkové soustavě vypadají dobře (třeba „50000000“) v logických 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 mínus 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;
```

## 2.12 Klopné obvody, registry a další...

Když už jsme téma synchronních obvodů načali, a teď jsme si tu i zavedli koncept časového signálu, pojďme se podívat na některé základní klopné obvody. Zájemce o další podrobnosti odkážu na knihu Hradla, volty, jednočipy.

*Magická písmena „FF“, která se v anglické literatuře v souvislosti s klopnými obvody používají, jsou zkratkou pro libozvučné označení klopného obvodu v angličtině: „Flip-Flop“. České označení „klopný obvod“, nezlobte se na mne, nezní tak hezky.*

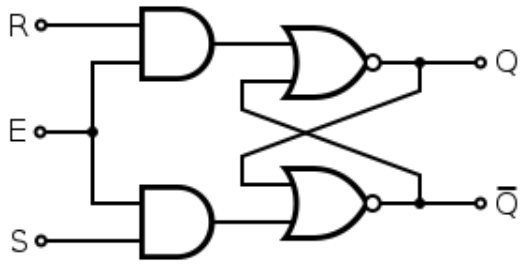
### Klopný obvod R-S

Nejjednodušší klopný obvod s dvěma asynchronními vstupy R a S a výstupem Q (u všech klopných obvodů bývá k dispozici i negovaný výstup /Q). Poskládáte si ho ze dvou hradel NAND či NOR (v reálném světě klidně 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 rsff 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;
```

### Klopný obvod R-S s povolovacím vstupem

K předchozímu typu přidáme 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 rseff 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;
```

## Klopný obvod 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).

*Podle popisu jde o DFF řízený úrovní (tedy něco jako obvod 7475). Ovšem my budeme chtít DFF řízený hranou, jako obvod 7474.*

Když přijde vzestupná hrana signálu E, zkopíruje se na výstup Q hodnota vstupu D (Data). V ostatních případech si výstup Q udržuje předcházející stav bez ohledu na vstupy.

```
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;
```

### Klopný obvod 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.

Pokud přivedeme 1 na vstup R, zapíše se hodnota ‘0’, pokud na vstup S, zapíše se hodnota ‘1’. Prioritu má vstup R.

```
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;
```

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);
```

### Dělička

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

*V praxi takto nikdo děličku syntetizovat nebude, místo toho použije proces, který s každou náběžnou hranou na vstupu přepne hodnotu na výstupu.*

### Klopný obvod 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 <= not Q_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.

### 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 „povolovacího 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 tu pomocný signál a samotná jeho změna se na stavu obvodu nijak neprojeví.

### Osmibitový posuvný registr (FIFO) se synchronním vstupem

Posuvné registry slouží k serializaci a deserializaci dat. Někdy se označují anglickou zkratkou FIFO – First In, First Out, čímž chce básník říct, že informace vystupují v takovém pořadí, v jakém vstupovaly. 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 zkopírována hodnota Din. Pokud je při náběžné hraně signál Load = '1', tak se do registru zkopíruje 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 tabulku:

```
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;
```

Pokud si pamatujete na úkol z jedné z minulých kapitol, totiž vytvořit dekodér pro sedmisegmentové LED, zde je ložená nápověda.

### 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;
```

### Šetříme...

Víte, co je to ALU? Je to anglické označení pro Aritmeticko-Logickou Jednotku, ve starší české literatuře tedy ALJ. Je to kombinační obvod, který má dva vícebitové vstupy, vícebitový výstup a několikabitový řídicí vstup. Ten říká, co se má s oběma operandy udělat, třeba sečíst nebo udělat operaci NAND. Ne, nebojte se, nebudeme si ukazovat, jak sestavit aritmetickou jednotku, to až později. Teď si ukážeme jen takovou 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 |
| **11** | G+H |

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

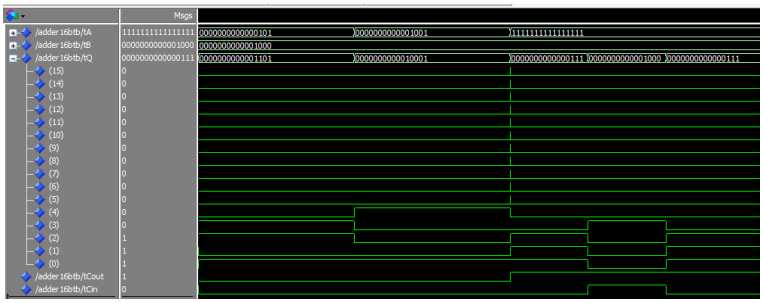
Obecně je dobrý zvyk co nejvíc sdílet právě aritmetické operace, porovnávání a podobné náročnější věci. Velmi snadno se stane, že v kódu napíšeme třeba velmi podobná sčítání do různých větví podmíněných příkazů (například ve stavovém automatu). Řešení je oddělit procesy a vyhradit si jeden speciální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

### Vylepšená sčítačka s předvídáním přenosu

Přiznávám se, chtěl jsem trochu mlžit, že *předvídání přenosu* je jakási magie, která umožňuje moderním obvodům, které pracují už trošku na hranicích kvantové fyziky, předvídat, co nastane. Ale realita je prostší a o hodně nudnější...

Když se podíváte na návrh naší plné šestnáctibitové sčítačky, všimnete si, že signál přenosu (carry) se proplétá celým obvodem od nejnižšího bitu k nejvyššímu. Ostatní signály jdou víceméně přímo, až na přenos. Ten se v každém bitu spočítá pro bezprostředně vyšší bit.

V ideálním světě to není problém, ale nežijeme v ideálním světě a ani elektronika není ideální. Ačkoli jsme to až dosud mohli úspěšně zanedbávat, tak i ve FPGA mají obvody určité nenulové zpoždění. Při takovém zpracování signálů, kdy každý prochází zhruba stejným počtem hradel, to až tak nevadí, ale postupně počítaný přenos (v anglické literatuře označovaný jako Ripple Carry) má tu protivnou vlastnost, že každý další stupeň má na výstupu platnou hodnotu až poté, co signál přenosu prošel přes všechny předchozí stupně. U šestnáctibitové sčítačky už i emulátor dokáže zaznamenat neplatné stavy, které vznikají v důsledku zpoždění – šesnáctý bit už je na výstupu sečtený, ale špatně, protože ještě nedorazil přenos z 15. bitu, kde se čeká na přenos ze 14. bitu atd.




Řešení tohoto problému nabízí právě technika „předvídání přenosu“ (někdy se setkáte i s označením „předpovídání“ či „predikce“, popřípadě s anglickou zkratkou CLA – „Carry Lookahead Adder“). Namísto postupného počítání z předchozích hodnot používá složitější výrazy (srovnatelné s vícevstupovými hradly), které se vyhodnocují najednou.

Trocha teorie: při sčítání dvou hodnot A a B (obecně, nejen u binárních čísel) můžeme definovat dva druhy přenosu: generovaný a šířený (*generated* a *propagated*).

**Generovaný přenos** na nějaké pozici vzniká tehdy (a jen tehdy), pokud oba sčítance mají na stejné pozici čísla, která při sečtení vyvolají přenos bez ohledu na to, zda přijde nebo nepřijde přenos z nižších řádů. Například při desítkovém sčítání čísel 43 a 82 vzniká na pozici desítek *generovaný přenos* – protože 4 + 8 vyvolá přenos do vyššího řádu bez ohledu na to, jak dopadne sčítání v nižším řádu.

**Šířený (propagovaný) přenos** na dané pozici vzniká tehdy, pokud oba sčítance mají na stejné pozici čísla, která při sečtení mohou vyvolat přenos, přijde-li z nižšího řádu. Pokud budeme sčítat 54 a 47, vznikne na pozici desítek *šířený přenos* – 5 + 4 dá 9, a pokud přijde přenos z nižšího řádu (tady přijde), tak se bude šířit dál.

U binárního sčítání vzniká generovaný přenos tehdy, pokud bity A i B jsou oba rovny 1. Generovaný přenos je roven výrazu A AND B, respektive bitově:

$G_N = A_N \text{ AND } B_N$

Šířený přenos může nastat jen tehdy, když je alespoň jeden ze sčítaných bitů jedničkový:

$P_N = A_N \text{ OR } B_N$

Výsledný přenos tedy vznikne buď jako generovaný (vždy), nebo jako šířený (pokud je přenos z nižšího řádu). V tabulce jsou shrnuty možné kombinace vstupních bitů A, B a vstupního pře-

nosu Ci, a k tomu výsledný přenos a příznaky P a G:

| A | B | Ci | Co | P | G |
|---|---|---|---|---|---|
| **0** | **0** | **0** | 0 | 0 | 0 |
| **0** | **0** | **1** | 0 | 0 | 0 |
| **0** | **1** | **0** | 0 | 1 | 0 |
| **0** | **1** | **1** | 1 | 1 | 0 |
| **1** | **0** | **0** | 0 | 1 | 0 |
| **1** | **0** | **1** | 1 | 1 | 0 |
| **1** | **1** | **0** | 1 | 1 | 1 |
| **1** | **1** | **1** | 1 | 1 | 1 |

Výstupní přenos je tedy roven výrazu G OR (P AND Ci).

*Někdy se P nedefinuje jako součet (OR), ale jako nonekvivalence (XOR). Rozdíl mezi OR a XOR je tu jen v případě A=B=1, ale v takovém případě vstupuje do hry generovaný přenos, takže to výsledek neovlivní. Výhoda je, že funkci XOR stejně sčítačka počítá – je to bit výsledku Q, který tedy můžeme použít místo P.*

U čtyřbitové sčítačky můžeme zapsat jednotlivé bity přenosu takto:

$C_0 = G_0$ OR ($P_0$ AND $C_{in}$),

$C_1 = G_1$ OR ($P_1$ AND $C_0$),

$C_2 = G_2$ OR ($P_2$ AND $C_1$),

$C_3 = G_3$ OR ($P_3$ AND $C_2$)

Což je vlastně popis sčítačky s postupným šířením přenosu – každý bit přenosu je závislý na hodnotě předchozího bitu. Ale můžeme substituovat:

$C_0 = G_0$ OR ($P_0$ AND $C_{in}$),

*$C_1 = G_1$ OR ($P_1$ AND ($G_0$ OR ($P_0$ AND $C_{in}$)))*, po rozvinutí dostaneme:

$C_1 = G_1$ OR ($P_1$ AND $G_0$) OR ($P_1$ AND $P_0$ AND $C_{in}$),

$C_2 = G_2$ OR ($P_2$ AND $G_1$) OR ($P_2$ AND $P_1$ AND $G_0$) OR ($P_2$ AND $P_1$ AND $P_0$ AND $C_{in}$),

$C_3 = G_3$ OR ($P_3$ AND $G_2$) OR ($P_3$ AND $P_2$ AND $G_1$) OR ($P_3$ AND $P_2$ AND $P_1$ AND $G_0$) OR ($P_3$ AND $P_2$ AND $P_1$ AND $P_0$ AND $C_{in}$)

Každý přenos se počítá z hodnot P a G, které jsou spočítané jedním hradlem, a hodnoty $C_{in}$, která do obvodu vstupuje a je konstantní. U šestnáctibitové sčítačky se tedy nečeká na přenos z bitu 15, který čeká na přenos z bitu 14, ...

Nevýhoda předvídání přenosu je, že poměrně brzy naroste složitost výrazů a počet nutných hradel. Proto se v praxi používá předvídání přenosů pro čtyřbitové nebo osmibitové sčítance a např. 32bitová sčítačka se řeší jako kaskáda čtyř osmibitových s předvídáním přenosu.

U FPGA naštěstí nebývá problém s „mnoha vícevstupovými hradly“ – logika je napevno nastavená v LUT – a problematická je pouze složitost výrazů (ne pro překladač, ale pro člověka). Naštěstí můžeme při konstrukci některé repetitivní prvky obejít pomocí konstrukce **for...generate**:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CLAdder8 is
  Port (
    A : in  STD_LOGIC_VECTOR (7 downto 0);
    B : in  STD_LOGIC_VECTOR (7 downto 0);
    Cin : in  STD_LOGIC;
    Q : out  STD_LOGIC_VECTOR (7 downto 0);
    Cout : out  STD_LOGIC);
end entity CLAdder8;

architecture gen of CLAdder8 is

signal G:STD_LOGIC_VECTOR (7 downto 0);
signal P:STD_LOGIC_VECTOR (7 downto 0);
signal C:STD_LOGIC_VECTOR (7 downto 0);
```

```
begin
  carry_loop: for i in 0 to 7 generate
    P(i) <= A(i) xor B(i);
    G(i) <= A(i) and B(i);
  end generate;

  Q(0) <= P(0) xor Cin;

  result_loop: for i in 0 to 6 generate
    Q(i+1) <= P(i+1) xor C(i);
  end generate;
  C(0) <= G(0) or (Cin and P(0));
  C(1) <= G(1) or (G(0) and P(1)) or
          (P(1) and P(0) and Cin);
  C(2) <= G(2) or (G(1) and P(2)) or
                  (G(0) and P(2) and P(1)) or
          (P(2) and P(1) and P(0) and Cin);

-- Vynecháme postupný rozvoj

  Cout <=
    G(7) or
   (G(6) and P(7)) or
   (G(5) and P(7) and P(6)) or
   (G(4) and P(7) and P(6) and P(5)) or
   (G(3) and P(7) and P(6) and P(5) and P(4)) or
   (G(2) and P(7) and P(6) and P(5) and P(4) and P(3)) or
   (G(1) and P(7) and P(6) and P(5) and P(4) and P(3) and P(2)) or
   (G(0) and P(7) and P(6) and P(5) and P(4) and P(3) and P(2) and P(1)) or
   (P(7) and P(6) and P(5) and P(4) and P(3) and P(2) and P(1) and P(0) and Cin);


end architecture;
```

S rozvojem výrazů pro přenos automatika nepomůže, ale všimněte si, jak konstrukce FOR pomohla vyřešit generování výrazů pro přenosy P a G a pro výstupní součet Q. Využil jsem výše popsaného triku, P generuji nikoli jako OR, ale jako XOR, a tím získávám zároveň mezisoučet, který stačí XORovat se vstupujícím přenosem.

## Inkrementor

Po vylepšené sčítačce si ukažme sčítačku lehce degradovanou. Degradujeme její funkci na prosté přičítání jedničky, což je činnost, která má řadu využití. Jak to udělat?

Můžeme použít plnou sčítačku, přivést do ní pouze jednu vstupní hodnotu, na druhý vstup pak přivést vektor, rovný jedné (00000001 např.) a vynulovat vstup Cin. Popřípadě obráceně: hodnotu přivést na vstup A, na vstup B přivést samé nuly, a jedničku připojit na vstup Cin.

Podívejme se ale na to, jak je postavená plná sčítačka. Jsou to dvě poloviční sčítačky za sebou, kde první dělá A+B=>P, druhá pak P+C=>Q. Je evidentní, že u inkrementoru nepotřebujeme dvě poloviční sčítačky, stačí nám jen jedna: A+B=>Q, kde v roli sčítance B bude přenos z nižšího řádu, popř. u nejnižšího řádu 1.

A protože víme, že $C_{OUT}$ = A AND B a Q = A XOR B, můžeme zapsat inkrementor tak, že:

- $C_0 = A_0$ AND $C_{in}$
- $C_{N+1} = A_{N+1}$ AND $C_N$
- $Q_0 = A_0$ XOR $C_{in}$
- $Q_{N+1} = A_{N+1}$ XOR $C_N$

Vytvořit generický inkrementor je teď už snadné. Jako cvičení si zkuste takto „degradovat“ sčítačku s předvídáním přenosu a navrhnout, jak by vypadal čítač, postavený z registru a inkrementoru...

## Násobička

Většina obvodů FPGA má v sobě zabudovanou hardwarovou násobičku, často ve více exemplářích. Například obvod Cyclone II, který používáme (EP2C5), má takových násobiček celkem 13, a to osmnáctibitových (na vstupu, tj. výstup o šířce 36 bitů). Kromě toho dokáží zkonstruovat až 26 násobiček 16x16 bitů pomocí logických obvodů a LUT.

Použití takové násobičky ve VHDL není žádné kouzlo – můžeme použít předpřipravený plug-in, který si vytvoříme pomocí „Megawizard Plug-In Manageru“ (ještě si jej představíme, násobičku tam najdete jako LPM_MULTIPLIER). Průvodce dá na výběr, kolik bitů mají mít operandy, jestli jeden z operandů nemá být konstantní, a můžete určit i to, jestli se mají využít dedikované obvody, případně logické elementy, nebo zda to necháte na syntetizéru. Pokud zadáte šířku větší, než má použitý obvod k dispozici, průvodce vygeneruje strukturu takovou, která

to zohlední, použije víc násobiček apod.

Druhý způsob je nechat to naprosto na uvážení syntetizéru a použít operátor násobení:

```
architecture main of multiplicator is
    signal A: signed(WORD_SIZE - 1 downto 0);
    signal B: signed(WORD_SIZE - 1 downto 0);
    signal Q : signed((WORD_SIZE * 2) - 1 downto 0);
begin
    Q <= A * B;
end architecture;
```

V takovém případě se při překladu rozhodne, zda použít dedikovanou násobičku, nebo jiné řešení.

## 2.13 Funkce, procedury, balíčky

Zase nastal čas se posunout od drátů k trošku vyšším abstrakcím.

### Funkce

Možná vás napadlo, že by bylo dobré některé opakované operace v procesech nadefinovat nějak obecněji, tak, aby byly znovupoužitelné, abyste nemuseli psát kód copy-and-paste, což je vždycky cesta do pekel. Kdyby tak VHDL mělo funkce, že? 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 dovoleno), 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 zkontrolovat. 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 zkontrolovat, 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 přetížíme 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_std z knihovny ieee, a z něj vezmi všechno“. Zjednodušuje to a usnadňuje znovupoužitelnost 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.

## Soubory

VHDL obsahuje i knihovnu pro práci se soubory. Samozřejmě nejde o způsob, jak naučit FPGA pracovat se soubory na disku, to tak snadno nejde. Soubory využijete především při testování, ať už pro zápis naměřených hodnot, nebo pro čtení testovací sady.

Práce se soubory je deklarovaná v knihovně TextIO (*use std.textio.all*). Tato knihovna definuje

typy LINE (řádek) a FILE (pole řádků), a k nim sadu základních operací read a write. Ale popořadě:

### Zápis do souboru

Začněme tím jednodušším, zápisem do souboru. Použijeme třeba testbench pro sčítačku adder_tb, jak jsme si jej zapsali v kapitole o GHDL. Pro připomenutí:

```
testing: process

  procedure vypis is
    begin
      report std_logic'image(tA) & " + " & std_logic'image(tB) & " = " & std_
logic'image(tCout) & std_logic'image(tQ);
  end procedure;

  begin
    tA <= '0'; tB <= '0'; wait for 10 ns;
    vypis;
    -- další testy, nebudu opakovat…
    report "Test OK";
    wait;
end process;
```

Procedura *vypis* vypisuje na konzoli kombinace bitů na vstupech a na výstupech. Jednoduchým postupem zajistíme, aby se data zapisovala do souboru. Nejprve si jej nadefinujeme:

```
testing: process

FILE test_log : text OPEN write_mode IS "adder_log.txt";
    VARIABLE row : line;

  procedure vypis is
-- pokračování…
```

Nadeklarovali jsme si soubor *test_log* typu TEXT, otevřený pro zápis (OPEN write_mode) a řekli jsme, jak se bude jmenovat. Zároveň jsme si nadeklarovali i proměnnou *row*, v níž si poskládáme řádek k zápisu.

Teorie práce se soubory je jednoduchá. Pomocí funkce write() zapíšeme potřebná data do řádku, a jakmile je řádek zaplněn, zapíšeme ho do souboru funkcí writeline():

```
procedure vypis is
  begin
    write(row, tA, left, 2);
    write(row, tB, left, 2);
    write(row, tCout, left, 2);
    write(row, tQ, left, 2);
    writeline (test_log, row);
end procedure;
```

*Při testování v GHDL nezapomeňte nastavit přepínače --fsynopsys a --std=08.*

Procedura write() má jako první parametr proměnnou, v níž se skládá řádek (row), pak hodnotu k výpisu (téměř libovolného typu), třetí parametr je zarovnání dat v rámci sloupce (left / right) a čtvrtý parametr je šířka sloupce (počet znaků).

Procedura writeline má jako první parametr soubor, jako druhý řádek, který je třeba zapsat.

K proceduře write existuje několik aliasů, které usnadňují práci se specifickými typy dat: swrite() pro výpis řetězce, hwrite(), bwrite() a owrite() pro výpis hodnoty hexadecimálně, oktalově a binárně. Parametry zůstávají stejné.

### Čtení ze souboru

Čtení ze souboru je analogické zápisu. Definice souboru i „přístupového řádku“ zůstává stejná, jen se použije „OPEN read_mode“.

Nejprve je potřeba ověřit, zda v souboru jsou ještě nepřečtené řádky, nebo zda jste na konci:

```
IF (NOT endfile(soubor)) THEN
      readline(soubor, row);
 ELSE
   WAIT;
END IF;
```

Tímto postupem načtete aktuální řádek do proměnné row. Z ní jednotlivé hodnoty přečtete do proměnných pomocí procedury read(). Hodnoty na jednom řádku musí být v textovém souboru odděleny mezerou nebo tabelátorem.

Procedura read() musí mít dva nebo tři parametry. První je načtený řádek, druhý je proměnná

(nelze načítat rovnou do signálu, musí být mezikrok s proměnnou), a třetí parametr udává, jestli byla hodnota načtena v pořádku (true/false). Třetí parametr není povinný.

Podobně jako u write() je i u read() několik aliasů: sread() pro čtení řetězce, bread(), oread() a hread() pro načítání binárních, osmičkových a hexadecimálních hodnot.

Pomocí čtení souboru můžeme například zautomatizovat test čtyřbitové sčítačky. Připravíme si testovací soubor (*adder4b_stim.txt* – obsahuje data, kterými budeme budit, tedy *stimulovat* testovanou komponentu, proto „stimulační soubor"), který bude obsahovat sadu testovacích kombinací – jeden sčítanec, druhý, vstupní přenos a výstupní hodnota:

```
0000 0000 0 00000
0001 0000 0 00001
1111 0000 0 01111
1111 0001 0 10000
0000 0000 1 00001
0001 0000 1 00010
1111 0000 1 10000
1111 0001 1 10001
```

Upravme testovací skript pro čtyřbitovou sčítačku. V proceduře testing si nejprve deklarujeme soubor, přístupovou proměnnou pro řádek a proměnné pro jednotlivé hodnoty:

```
FILE test_stimul : text OPEN read_mode IS "adder4b_stim.txt";
VARIABLE row : line;
VARIABLE inp1, inp2 : std_logic_vector (3 DOWNTO 0);
VARIABLE result : std_logic_vector (4 DOWNTO 0);
VARIABLE carryIn : std_logic;
```

Samotná procedura začne tak, jak jsme si popsali výše – načtením jednoho řádku. Pak načteme jednotlivé hodnoty do proměnných, přiřadí je k signálům, chvilku počkáme a zkontrolujeme, zda výsledek odpovídá očekávanému (použijeme assert):

```
  IF (NOT endfile(test_stimul)) THEN
      readline(test_stimul, row);
    ELSE
      WAIT;
    END IF;
    read(row, inp1);
    read(row, inp2);
    read(row, carryIn);
```

```
    read(row, result);

    tA <= inp1;
    tB <= inp2;
    tC <= carryIn;
    WAIT FOR 10 ns;
    ASSERT (tCout & tQ) = result REPORT "failed" SEVERITY failure;
```

Testbench si tak automaticky přečte data, na kterých má komponentu otestovat, a provede jeden test po druhém.

Můžeme oba přístupy spojit, z jednoho souboru data číst a do druhého zapisovat požadovaný výsledek, reálný výsledek a informaci o tom, jestli daný test proběhl dobře. Celý testbench pak vypadá takto:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE std.textio.ALL;

ENTITY adder4b_tb IS
END;

ARCHITECTURE bench OF adder4b_tb IS
  COMPONENT adder4B 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 COMPONENT;

  SIGNAL tA, tB, tQ : std_logic_vector (3 DOWNTO 0);
  SIGNAL tC, tCout : STD_LOGIC;

BEGIN
  testing : PROCESS

FILE test_stimul : text OPEN read_mode IS "adder4b_stim.txt";
    VARIABLE row : line;
    VARIABLE inp1, inp2 : std_logic_vector (3 DOWNTO 0);
```

```
    VARIABLE result : std_logic_vector (4 DOWNTO 0);
    VARIABLE carryIn : std_logic;

FILE test_log : text OPEN write_mode IS "adder4b_log.txt";
    VARIABLE orow : line;

  BEGIN

    IF (NOT endfile(test_stimul)) THEN
      readline(test_stimul, row);
    ELSE
      WAIT;
    END IF;

    read(row, inp1);
    read(row, inp2);
    read(row, carryIn);
    read(row, result);

    -- kopie vstupních dat
    write(orow, inp1, left, 5);
    write(orow, inp2, left, 5);
    write(orow, carryIn, left, 2);
    write(orow, result, left, 6);
    tA <= inp1;
    tB <= inp2;
    tC <= carryIn;
    WAIT FOR 10 ns;

    -- zapíšeme výsledek
    write(orow, (tCout & tQ), left, 6);

    -- a ještě informace o tom, zda vše proběhlo jak mělo
    IF (tCout & tQ) = result THEN
      swrite(orow, "OK", left, 2);
    ELSE
      swrite(orow, "FAIL", left, 4);
    END IF;
    writeline(test_log, orow);
    --vypis;
  END PROCESS;
```

```
  UUT : adder4B PORT MAP(tA, tB, tC, tQ, tCout);

END bench;
```

## 2.14 VHDL 2008

Ani v elektronice se čas nezastavil. I jazyk VHDL se vyvíjí. Jeho první standard vydalo sdružení IEEE v roce 1987. Následovalo několik revizí, a ta poslední nese označení VHDL 2008 (přesněji VHDL 1076-2008).

Verze 2008 přinesla některé novinky, které jsme si popisovali, jako jsou například vylepšené bitové řetězce s udáním typu:

```
data <= X"00"; -- hexadecimální osmibitové číslo
```

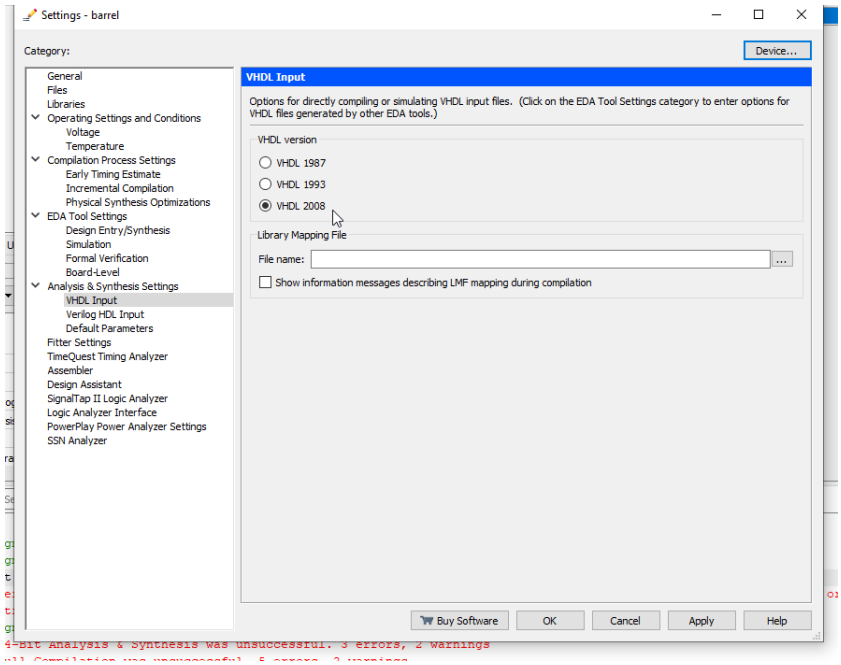
Další novinky jsou například podmíněné generování (if..elsif..else nebo case), použití slova „all“ v senzitivity listu (proces je citlivý na změny všech signálů, které se uvnitř něho čtou). Nové je i rozšíření možnosti generických parametrů, jimiž lze předávat třeba i datové typy.

V této verzi můžete například při testování „vnutit“ hodnotu signálu slovem „force“, nebo jej opět uvolnit pomocí „release“:

```
v <= force in '1';  -- hodnota vnucená na vstup
v <= force out '0'; -- hodnota vnucená na výstup

v <= release in;    -- uvolnění vstupu
v <= release out;   -- uvolnění výstupu
```

A v neposlední řadě přineslo VHDL 2008 konečně i blokové víceřádkové komentáře /* ... */

*Upozornění – může se vám stát, že vám vývojové prostředí Quartus zahlásí, že určitá konstrukce není povolena (například víceřádkové blokové komentáře). Překladači totiž musíte explicitně říct, aby používal verzi VHDL 2008, a to v menu Assignments – Settings, položka Analysis and Synthesis Settings – VHDL Input. Zde bývá defaultně nastavená verze VHDL 1993.*


Settings - barrel
Category:
Device...
General
Files
Libraries
Operating Settings and Conditions
Voltage
Temperature
Compilation Process Settings
Early Timing Estimate
Incremental Compilation
Physical Synthesis Optimizations
EDA Tool Settings
Design Entry/Synthesis
Simulation
Formal Verification
Board-Level
Analysis & Synthesis Settings
VHDL Input
Verilog HDL Input
Default Parameters
Fitter Settings
TimeQuest Timing Analyzer
Assembler
Design Assistant
SignalTap II Logic Analyzer
Logic Analyzer Interface
PowerPlay Power Analyzer Settings
SSN Analyzer
VHDL Input
Options for directly compiling or simulating VHDL input files. (Click on the EDA Tool Settings category to enter options for VHDL files generated by other EDA tools.)
VHDL version
VHDL 1987
VHDL 1993
VHDL 2008
Library Mapping File
File name:
Show information messages describing LMF mapping during compilation
Buy Software
OK
Cancel
Apply
Help
4-bit Analysis & Synthesis was unsuccessful. 3 errors, 2 warnings

# 3 Podrobněji o FPGA

# 3 Podrobněji o FPGA

## 3.1 Jak FPGA pracují?

Opakování je matka moudrosti, a proto si připomeňme, že FPGA neobsahují „volná hradla" a „magické vodiče", ale poněkud sofistikovanější stavební kameny, totiž „logické buňky" (logic cell), kterých jsou na čipu desítky tisíc (u těch nejmenších) až jednotky milionů (Intel v čipu Stratix 10 GX 10M dosahuje velikosti přes 10 milionů logických buněk).

Nejjednodušší logická buňka obsahuje programovatelnou LUT (Look-Up Table), klopný obvod a multiplexor, který umožňuje přemostit právě klopný obvod. LUT mívají typicky málo bitů, např. 4, což umožňuje snadné zapsání jakékoli logické funkce pro až čtyři vstupy.

Mezi logickými elementy je „glue logic", *temná hmota z multiplexorů*, která propojuje jednotlivé elementy podle zadané konfigurace. Každý logický element se může propojit s několika sousedními – s kterými a jak, to je zadáno v konfiguračním souboru, podle kterého se nastavují multiplexory. Můžeme si to představit asi jako vjezd či výjezd z nádraží, kde se třeba deset kolejí sbíhá do dvou a naopak.

Kromě logických elementů a *propojovací hmoty* jsou na čipu integrované i takzvané *hraniční elementy* (boundary elements, IO cells), které obsluhují jednotlivé piny FPGA. Obsahují proto nastavitelné napěťové budiče (můžete si u mnohých pinů vybrat, s jakou logikou pracují) a oddělovače.

Kromě univerzálně nastavitelných propojovacích cest existují ve FPGA dedikovaná propojení sousedních buněk. Nejběžnější typ takovéto propojovací cesty je „řetězec přenosu" (carry), s nímž jsme se setkali u sčítačky. Při její implementaci tak syntetizér může využít tyto cesty a zrychlit tím šíření signálu (obchází se multiplexory).

Kromě logických buněk a propojovacího systému mají FPGA integrovány i bloky statické paměti RAM – většinou jako „dual port" paměti, do nichž mohou přistupovat dva různé systémy naráz. Použití této paměti si ještě budeme ukazovat.

## 3.2 Piny a jejich přiřazení

Čipy FPGA jsou obří a mívají i několik set pinů. Některé z nich mají pevně určenou funkci (dedicated), ale naprostá většina je uživatelsky konfigurovatelná (user pins, IO pins).

Uživatelské piny jsou naprosto pod vaší kontrolou. Můžete si určit, zda jsou vstupní,

výstupní, nebo obousměrné s třístavovou logikou, případně s pull-up rezistorem. Každý pin je uvnitř FPGA připojený k hraniční buňce, která poskytuje potřebné buzení.

Dedikované piny mají napevno přiřazené funkce. Dělí se do tří kategorií:

- Napájení
- Systémové / konfigurační piny, po kterých probíhá nahrávání konfigurace
- Dedikované vstupy pro hodiny

Hodinové vstupy jsou speciálně ošetřené a rozvedené po velké ploše čipu tak, aby byly hodinové signály dobře dostupné.

Napájecí vstupy se rozlišují na napájení jádra (core voltage) a napájení rozhraní (IO voltage). Napájení jádra (VCC nebo VCCINT) je pevně dané podle typu konkrétního FPGA. U těch nejstarších to bylo 5 V, novější pracují s mnohem nižším napětím – zjednodušeně řečeno čím novější, tím nižší napětí, takže se jeho hodnota postupně snižovala z 3,3 V na 2,5 V, pak 1,8 V, 1,5 V a 1,2 V; nejnovější pak ještě nižší.

Napájení rozhraní (VCCO nebo VCCIO) je používané pro budiče v hraničních buňkách. Toto napětí by mělo odpovídat napětí, použitému ve zbytku systému.

Většina dnes používaných FPGA má piny rozdělené do několika „napěťových bank“, takže jako konstruktér můžete určit několik napěťových úrovní pro vstupy a výstupy a nechat tak část obvodu pracovat s periferiemi 3,3 V, jinou třeba na 1,8 V.

U většiny zapojení budete chtít určit, ke kterému pinu má být přiřazen jaký signál. Pokud to neuděláte, přiřadí je syntetizér, jak mu přijde nejvhodnější. Ale většinou pracujete s nějakými omezeními, třeba máte kit, kde jsou napevno připojené periferie, a tam nezbývá než říct, co chcete kam zapojit.

Vývojová prostředí pro tento účel mají nástroj, zvaný „pin planner“. Ten vám umožní vybrat přiřazení signálu pro hlavní entitu, určit, jak se má s pinem nakládat, jestli má být třístavový, kolika voltovou logiku má použít, někdy i maximální výstupní proud a řadu dalších parametrů.

Ve skutečnosti je přiřazení pinů obyčejný textový zápis typu:

```
set_location_assignment PIN_26 -to LED -- Intel / Altera
```

```
NET "LED" LOC = "P17"; -- Xilinx
```

Jeden z kroků syntézy je proces "P&R", tedy "place and route", během kterého se určují, které fyzické elementy budou opravdu použity a jak budou propojeny. Zde se právě zohledňuje přiřazení pinů. Po tomto kroku máte hotový binární kód, který lze nahrát do paměti konfigurace, popřípadě do čipu.

## 3.3 Hodinové signály

Hodiny jsou ve FPGA alfou a omegou. FPGA jsou synchronní zařízení, což znamená, že změna stavu klopných obvodů probíhá s náběžnou hranou hodinového signálu. To při návrhu sice ušetří spoustu problémů, ale jiné způsobí. Jedním z těch větších je nutnost vést hodinové pulsy tak, aby pokud možno ke každému klopnému obvodu dorazil puls ve stejný čas. Představte si, že by se hodiny šířily z levého horního rohu jako vlny na vodě – do pravého dolního by dorazily významně později, než byly v levém horním, a pokud by nějaké spojení vedlo „proti směru času", bylo by zle.

*Představte si čip s hranou 1 cm, což je na dnešní poměry docela malý čip. Světlo tuto vzdálenost uletí „téměř okamžitě", ale ne nekonečně rychle. Stačí chvilka počítání a zjistíte, že kdyby měl signál frekvenci 15 GHz a šířil se zleva doprava, tak na jednom okraji mají protichůdnou fázi hodin než na druhém. Reálně stačí mnohem menší frekvence, protože signál neprochází obvodem ani zdaleka rychlostí světla.*

Proto se pro šíření hodinového signálu používají speciální vyhrazené signály uvnitř čipu, které jsou vedené tak, aby zpoždění bylo minimální a klopné obvody přepínaly opravdu „téměř synchronně". Většina FPGA používá jeden nebo několik zdrojů hodinového kmitočtu, připojených na dedikovaný „hodinový vstup". To je důležité, protože pouze tyto vstupy jsou připojené ke „globálnímu rozvodu hodin".

Další problém s hodinovými signály spočívá v tom, že různé části systému mohou pracovat s různými hodinovými signály – takříkajíc *v jiném časovém pásmu*. Blíže o tomto problému budeme hovořit v kapitole „Hodinové domény".

## 3.4 Nahrávání konfigurace do kitu EP2C5

Když se podíváte na doporučený začátečnický kit s čipem EP2C5T144, uvidíte, že pro připojení programátoru má k dispozici dva konektory, jeden označený JTAG a druhý označený AS.

JTAG (Joint Test Action Group) je standard, definovaný IEEE. Standardizované jsou signály rozhraní, nikoli už jejich rozmístění na konektoru, a tak každý výrobce používá trochu jiné

zapojeni nebo jiný konektor. Bohužel. Pak máte několik různých JTAG zařízení s různými konektory (třeba jen pro ARM mám dva různé programátory).

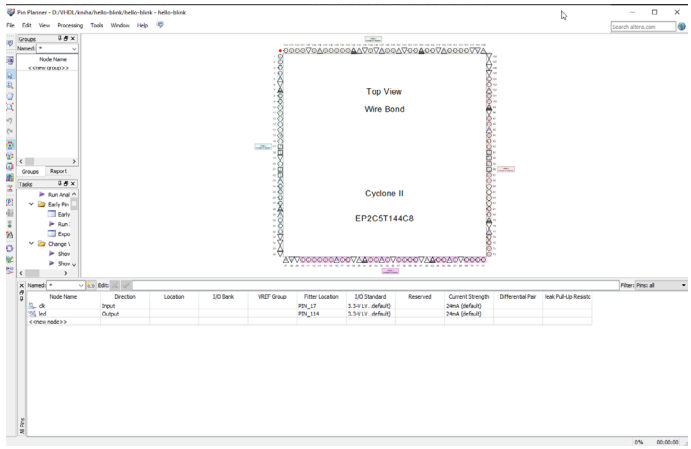
Pro FPGA od Altery / Intelu doporučuju jednoduchý levný čínský USB Blaster. Většina prodejců jej přibaluje ke kitu.


Připojte jej dodaným kabelem ke konektoru JTAG, USB kabelem do PC, a po instalaci ovladačů můžete začít programovat.

Před programováním kitu je potřeba udělat ještě jednu důležitou věc, totiž v Pin Planneru nastavit správně přiřazení pinů k portu hlavní entity.

Neobejdete se přitom bez znalosti skutečného zapojení kitu, tedy buď schématu, nebo alespoň přiřazení vývodů. Obojí naleznete v příloze. U tohoto kitu jde hlavně o piny 17 (hodinový vstup), 144 (tlačítko), 3, 7 a 9 (LED) a pin 73 (jednoduchý RC obvod, použitelný např. jako RESET).

Vraťme se k příkladu s blikačem. Hlavní entita má jeden vstup (hodiny 50 MHz) a jeden výstup (LED). Po ověření syntaktické správnosti je načase spustit Pin Planner:

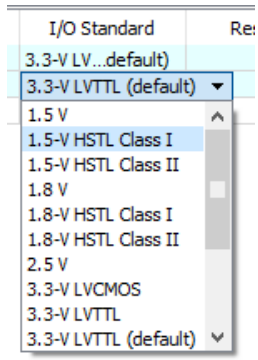


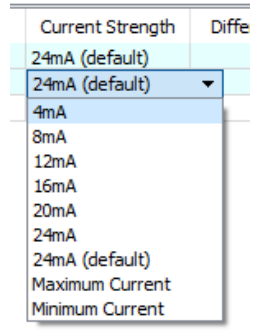

Všimněte si, že oba signály jsou připravené v seznamu, a dokonce mají nějaké vývody už přiřazené – syntetizér usoudil, jaké by byly vhodné. S pinem clk se trefil: správně usoudil, že jde o hodiny, a přiřadil mu dedikovaný hodinový vstup z pinu 17. Změníme tedy hlavně výstup LED:

Named: * Edit:

| Node Name | Direction | Location | I/O Bank | VREF Group | Fitter Location | I/O Standard | Reserved |
|---|---|---|---|---|---|---|---|
| clk | Input | PIN_17 | 1 | B1_N0 | PIN_17 | 3.3-V LV...default) | |
| led | Output | PIN_7 | 1 | B1_N0 | PIN_114 | 3.3-V LV...default) | |
| <<new node>> | | | | | | | |

Všimněte si údajů, o nichž jsme se bavili výše, například možnost nastavit pinu úroveň napětí nebo proudu, popřípadě zapnout pull-up:



Kompletní překlad (Processing – Start Compilation, popř. Ctrl+L) sestává z několika kroků. První je „analýza a syntéza“, pak fáze „P&R“ (Place and route, tedy umístění a propojení), následuje sestavení (assembler), kdy se generují programovací soubory SOF a POF (najdete je v podadresáři *output_files*). Poté proběhne ještě analýza časování (TimeQuest Timing Analysis) a příprava dat pro simulátor (EDA Netlist).

Jakmile je překlad u konce, můžeme programovat reálné zařízení. Připojte USB Blaster ke kitu (konektor JTAG) a připojte ke kitu napájení. Z menu Tools vyberte Programmer, nebo v okně Tasks klikněte na Program Device.

V programátoru nastavte rozhraní na JTAG a zvolte funkci Auto Detect. Programátor prozkoumá připojená zařízení (JTAG umožňuje připojit více zařízení do řetězce) a nalezne připojený kit EP2C5. Vyberte soubor (Add File) – pro programování přes JTAG bude zapotřebí soubor SOF.

Pak stačí kliknout na Start a během několika sekund se přenesou potřebná data do vnitřní konfigurační paměti FPGA. Pokud bylo vše v pořádku, LED začne blikat.

Rozhraní JTAG je určeno především pro komunikaci s „živým“ zařízením. Dokáže při provozu zjišťovat stav obvodu a pinů. Pomocí megafunkcí „JTAG Signal Tap Analyzer“ nebo „JTAG In-System Sources and Probes“ si můžete do obvodu přidat hotové komponenty, které fungují jako „virtuální logické sondy“. V reálném zařízení tak můžete sledovat průběhy signálů uvnitř FPGA, jako byste logickým analyzátorem sledovali signály v zařízení, poskládaném z integrovaných obvodů.

Jakmile kit odpojíte, konfigurace zmizí. Nezapomeňte – nahrávali jsme ji do vnitřní paměti FPGA, a ta je volatilní, tj. při odpojení napájení vše zapomene. Proto se k FPGA připojují speciální sériové FLASH, v nichž jsou uložena konfigurační data. Ta se po zapnutí napájení nahrají do paměti FPGA.

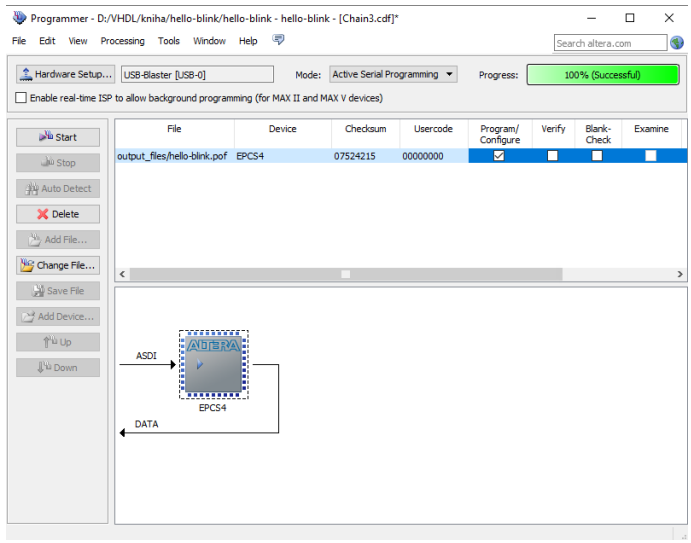
Otázka zní: Jak nahrajeme data do této paměti?

Slouží k tomu druhé rozhraní na kitu, nazvané AS (někdy ASP) – Active Serial. Když přepojíte USB Blaster k tomuto rozhraní, připojili jste jej přímo k paměti FLASH.

V programátoru teď musíte postupovat trochu jinak. V první řadě zvolíte mód práce „Active Serial Programming“. Přidáte zařízení (Add Device), ale tentokrát musíte vybrat ne typ FPGA, ale typ paměti. Na našem kitu to je paměť EPCS4 (4 Mbit FLASH, tedy 0,5MB).

Ve schématu v dolní polovině programátoru klikněte na symbol této paměti a zvolte možnost „Change file“. Vyberte soubor k naprogramování – tentokrát to bude POF. Zaškrtněte „Program / Configure“, tím oznámíte, že chcete paměť naprogramovat konfiguračním souborem. A pak

jen klikněte na Start.




Do paměti se nahraje soubor POF. Po skončení můžete kit vypnout. Když jej opět zapnete, nahraje se tento soubor do FPGA – a LED začne blikat.

### Když AS není k dispozici

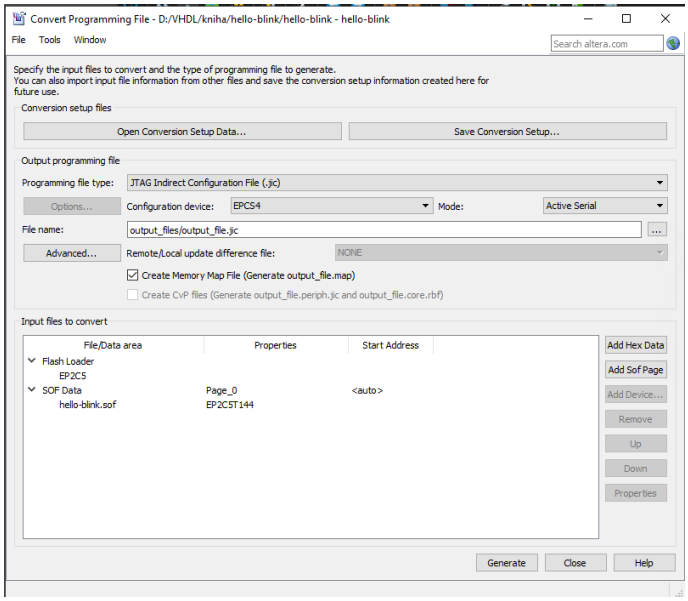
Jsou kity, na nichž port AS není. Jak v takovém případě postupovat?

Postup je založen na tom, že FPGA na chvíli proměníte v programátor sériových FLASH.

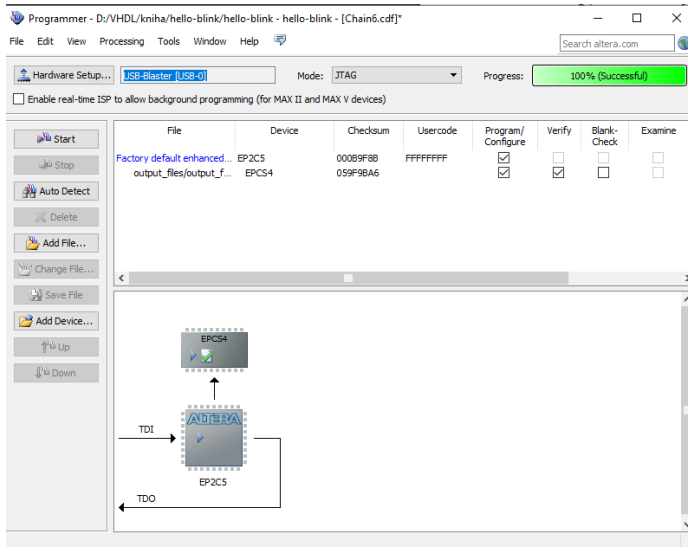
K naprogramování FLASH přes Serial Loader budete potřebovat speciální soubor JIC (JTAG Indirect Configuration) nebo JAM. Quartus naštěstí umožňuje takový soubor vygenerovat ze souboru SOF. Slouží k tomu konvertor, který najdete v menu File pod názvem Convert Programming Files. Postup je následující:

- Spusťte konvertor
- V dialogovém souboru vyberte v roletce „Programming file type“ typ „JTAG Indirect... (.jic)“
- V poli „Configuration device“ zvolte typ FLASH (u nás „EPCS4“).

- V poli „File name“ zadejte jméno vygenerovaného souboru.
- Ve spodní polovině okna klikněte na SOF Data.
- Přidejte soubor kliknutím na Add File a vybráním souboru SOF, který chcete nahrát
- Klikněte na položku Flash Loader
- Přidejte zařízení (Add Device) a vyberte typ FPGA (EP2C5)
- Spusťte převod tlačítkem Generate




Jakmile máte JIC připravený, otevřte programátor. V programátoru klikněte na „Add File“, ale nevybírejte SOF, ale vygenerovaný JIC. Schéma konfigurace se změní:




Schematicky je naznačeno, že se data nahrávají do FPGA, a pak přenášejí do FLASH. Zaškrtněte políčka „Program/Configure“, popřípadě i Verify. Po spuštění procesu kliknutím na Start nahraje programátor do FPGA konfiguraci s aktivním Serial Loaderem, který poslouží k naprogramování připojené FLASH. Jakmile proces doběhne, je hotovo. Konfigurace je natrvalo nahraná do FLASH.

# 4 Analogový výstup

# 4 Analogový výstup

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

Pokud je blikání LEDkou obdoba 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. FPGA jsou také většinou digitální (nové modely mívají i analogové subkomponenty, ale ty nechme stranou), tak budeme muset použít stejný způsob.

## 4.1 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 sítě, 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 nazvěme si ji D. K ní si uděláme čítač hodinových pulsů, rovněž osmibitový, který bude počítat od nuly k 255 a pak znovu 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. Průběh signálu pro hodnoty 1, 7 a 15 vypadá takto:

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 čítat 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 main 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;

  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 architecture;
```

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 rozsvěcení bude pohasínat. Nejjednodušší změna je nastavit v<=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 nestejné 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:

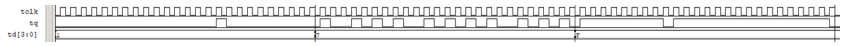
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

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 zkusíte 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;
```

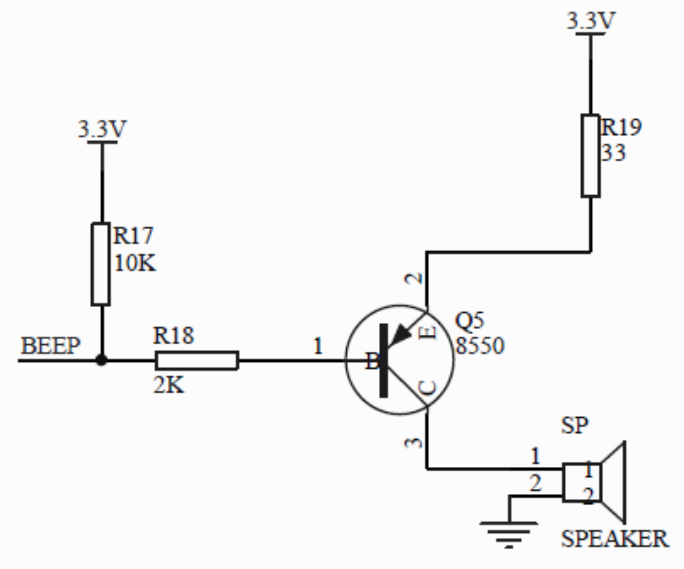
## 4.2 Pokus: FPGA siréna

Mezi blikáním LEDkou a generováním zvuku zas tak velký rozdíl není, jen ve frekvenci. Lidské oko dokáže rozpoznat blikání o maximální frekvenci zhruba 25 bliknutí za sekundu, rychlejší už splývá. Lidské ucho dokáže slyšet zvuky s frekvencí 16 Hz až 16.000 Hz – mladí lidé slyší i mnohem vyšší zvuky, až přes 20 kHz, s přibývajícími léty ale slyšitelný rozsah klesá.

Když tedy „zrychlíme blikání LED“ a na výstup místo LEDky připojíme nějaký zvukový měnič, dostaneme místo blikače bzučák.

*Na tomto místě varuju, že FPGA na výstupu nedává dostatečný proud k tomu, aby rozkmital membránu běžného reproduktoru. Můžete se setkat se zapojeními, kde se používají sluchátka s velkou impedancí, ale i tak chci před podobným postupem varovat: vždy zapojujte zátěž, v níž jsou cívky, přes oddělovací a budicí člen s tranzistorem. Ten jednak zvýší možnou proudovou zátěž, ale také ochrání digitální výstup před indukčními napěťovými rázy, které jej mohou zničit.*

Pro ukázku – takto je v kitu OMDAZZ připojen bzučák:




Zkuste si připojit takto jednoduše buzený reproduktor k některému vhodnému pinu FPGA a upravit blikač tak, aby výsledná frekvence byla třeba komorní A, tj. 440 Hz.

Pro hodinový kmitočet 50 MHz vychází dělicí konstanta rovna 113636. Když touto konstantou podělíme 50MHz hodiny, dostaneme výsledných 440 Hz (a nějaké ty setiny). Naštěstí to nemusíme počítat, necháme to na entitě:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY delicka IS
    GENERIC (
        fmain : INTEGER := 50_000_000;
        fout : INTEGER := 440
    );
    PORT (
        clk : IN std_logic;
        sound : OUT std_logic
    );
END ENTITY;

ARCHITECTURE main OF delicka IS

    CONSTANT divider : INTEGER := fmain / fout / 2;

BEGIN
    PROCESS (clk) IS
        VARIABLE counter : INTEGER := 0;
        VARIABLE blik : std_logic := '0';
    BEGIN
        IF (rising_edge(clk)) THEN
            counter := counter + 1;
            IF (counter = divider) THEN
                counter := 0;
                blik := NOT blik;
            END IF;
        END IF;
        sound <= blik;
    END PROCESS;
END ARCHITECTURE;
```

Všimněte si, že spočítaný poměr dělíme dvěma. Proč?

Kdybychom pracovali s dělitelem 113636, museli bychom zajistit, že než čítač napočítá do této hodnoty, tak proběhne **celá perioda** výstupního signálu, tedy přepnutí nahoru i dolů. Museli bychom tedy signál jednou přepnout při hodnotě 56818, jednou při 113636.

Pohodlnější je pracovat pouze s poloviční hodnotou, což je těch 56818, a při jejím dosažení jen přepnout výstupní signál. Proto počítáme dělicí konstantu jako $f_{\text{vstupní}} / f_{\text{výstupní}} / 2$.

Pokud použijeme dva generátory, každý s jinou výškou tónu, a třetí s velmi nízkým kmitočtem, třeba okolo 1 Hz (LFO – Low Frequency Oscillator), který bude mezi danými dvěma tóny přepínat, získáme při vhodném naladění hasičskou sirénu a známý signál „hoří“…

*Vhodné naladění znamená dva tóny, mezi nimiž je interval čisté kvarty, tedy například tóny E a A – můžete použít třeba frekvence 659 a 880, nebo 5274 a 7040...*

```
ENTITY sirena IS
    GENERIC (
        fmain : INTEGER := 50_000_000;
        f1 : INTEGER := 5274;
        f2 : INTEGER := 7040;
        fslow : INTEGER := 1
    );
    PORT (
        clk : IN std_logic;
        sound : OUT std_logic
    );
END ENTITY;

ARCHITECTURE main OF sirena IS

    SIGNAL s1, s2, sel : std_logic;

BEGIN

    tone1 : ENTITY work.delicka GENERIC MAP (fmain => fmain, fout => f1) PORT MAP (clk =>
clk, sound => s1);
    tone2 : ENTITY work.delicka GENERIC MAP (fmain => fmain, fout => f2) PORT MAP (clk =>
clk, sound => s2);
    lfo : ENTITY work.delicka GENERIC MAP (fmain => fmain, fout => fslow) PORT MAP (clk =>
clk, sound => sel);
```

```
    sound <= s1 WHEN sel = '0' ELSE
        s2;

END ARCHITECTURE;
```

Výsledný průběh může vypadat nějak takto (přizpůsobil jsem frekvence tak, aby změna byla patrná):


Můžete si všimnout toho, že vlivem nesoudělnosti kmitočtů a nezávislosti obou generátorů můžou vzniknout na výstupu pulsy s délkou, která neodpovídá ani jedné zdrojové frekvenci. U sirény to nevadí, ucho to nemá šanci poznat, ale někde jinde by to mohlo vadit. Řešení by mohlo být, že vždy necháme doběhnout aktivní cyklus a zdroj přepneme až se sestupnou hranou. Jenže tím nevyřešíme synchronizaci fází, tj. aby čítač v generátoru vždy v okamžiku přepnutí začínal od nuly.

Proto použijeme řešení jiné. Generickou děličku upravíme tak, že dělitel nebudeme zadávat jako parametr, ale budeme jej posílat jako vektor. Navíc s ním nebudeme pracovat přímo – budeme si udržovat vlastní kopii hodnoty, a vnější přepíšeme jen ve chvíli, kdy ukončíme celý výstupní cyklus (tedy *při sestupné hraně výstupního signálu*).

```
ENTITY delicka2 IS
    GENERIC (wide : INTEGER := 23);
    PORT (
        clk : IN std_logic;
        sound : OUT std_logic;
        div : IN unsigned (wide - 1 DOWNTO 0)
    );
END ENTITY;

ARCHITECTURE main OF delicka2 IS
    SIGNAL divider : unsigned (wide - 1 DOWNTO 0) := div;

BEGIN
    PROCESS (clk) IS
        VARIABLE counter : INTEGER := 0;
        VARIABLE blik : std_logic := '0';
    BEGIN
        IF (rising_edge(clk)) THEN
            counter := counter + 1;
```

```
            IF (to_unsigned(counter, wide) = divider) THEN
                counter := 0;
                blik := NOT blik;
                IF (blik = '0') THEN
                    divider <= div;
                END IF;
            END IF;
        END IF;
        sound <= blik;
    END PROCESS;
END ARCHITECTURE;
```

Jak vstupní port pro dělitel, tak interní signál „divider“ definujeme jako unsigned s danou šířkou. Šířka 23 bitů je dostatečná pro dělitele, které budeme používat.

Všimněte si popsaných změn, zejména toho, že hodnotu z portu div propisujeme do signálu divider ve chvíli, kdy se mění výstup (proměnná *blik*), a mění se na nulu. V tu chvíli je i interní čítač roven nule, takže změna hodnoty je bezpečná.

Práce s touto děličkou se trochu změní. Musíme zajistit posílání správné hodnoty dělitele.

```
ENTITY sirena2 IS
    GENERIC (
        fmain : INTEGER := 50_000_000;
        f1 : INTEGER := 5274;
        f2 : INTEGER := 7040;
        fslow : INTEGER := 1
    );
    PORT (
        clk : IN std_logic;
        sound : OUT std_logic
    );
END ENTITY;

ARCHITECTURE main OF sirena2 IS

    CONSTANT wide : INTEGER := 23;

    SIGNAL sel : std_logic := '0';
    SIGNAL div : unsigned (wide - 1 DOWNTO 0) := to_unsigned(1, wide);
```

```
BEGIN

    tone : ENTITY work.delicka2 GENERIC MAP (wide => wide) PORT MAP (clk => clk, sound =>
sound, div => div);
    lfo : ENTITY work.delicka GENERIC MAP (fmain => fmain, fout => fslow) PORT MAP (clk =>
clk, sound => sel);

    div <= to_unsigned(fmain/f1/2, wide) WHEN sel = '0' ELSE
        to_unsigned(fmain/f2/2, wide);

END ARCHITECTURE;
```

Všimněte si, že jsme v roli LFO použili původní děličku – šla by samozřejmě použít nová, ale není to nutné. Podle výstupu z LFO se mění hodnota signálu *div* – zde tedy nově počítáme dělitel ze známých frekvencí.

Výsledek je již bez parazitních frekvencí:


Na druhou stranu změna frekvence tónu nepřichází okamžitě se změnou LFO. Ale jako u spousty jiných situací i zde platí, že *obojí mít nemůžete*...

A za domácí úkol si zkuste upravit děličku tak, aby na výstup neposílala hodnoty 0 a 1, ale třeba 0 a 255. Výstup pak posílejte ven přes osmibitový převodník sigma-delta.

Máte to? Zkuste ještě děličku upravit tak, aby místo obdélníku posílala třeba pilovitý průběh 0-255.

*Tip: celý cyklus si rozdělte na 256 kroků. Dělitel bude 256x menší a s každým dopočítáním do maxima zvětšíte interní čítač kroku o 1. U pilovitého průběhu je pak výsledná hodnota rovna aktuálnímu kroku, u obdélníkového průběhu můžete na výstup pouštět hodnotu nejvyššího bitu kroku apod. Pomocí tabulky konstant můžete dokonce přepočítat tuto osmibitovou hodnotu na sinus.*

Můžete využít techniku DDS (digitální syntéza) a uložit si například tabulku hodnot pro funkci sinus tak, že vstupem bude osmibitová „adresa samplu“ a na výstupu osmibitová (nebo kolikabitová chcete) hodnota předpočítané funkce sinus. Pokud na vstup pošlete pilovitý signál, na výstupu získáte sinusoidu. Ukázka je třeba zde:

https://surf-vhdl.com/how-to-generate-sine-samples-in-vhdl/

U osmibitové děličky bude tedy maximální možná frekvence rovna 193 kHz (50 MHz / 256). Pokud chcete vyšší frekvenci, použijte metodu, při níž dělička neprojde nutně všemi hodnotami a některé vynechá.

*Připomínám, že zdrojové kódy příkladů i cvičení najdete na https://datacipy.cz*

# 5 Paměti

# 5 Paměti

Celé tajemství digitální paměti je v elementech, které uchovávají svoji hodnotu. Kdo tipuje, že to jsou klopné obvody, má pravdu. Ve FPGA nevyužijeme další možnosti (kapacita, magnetické pole atd.), takže budeme odkázáni jen na ně. Pojďme si je ještě jednou připomenout od A do Z – vlastně „od D do T“. A taky si řekneme, jak ve VHDL vytvořit opravdovou paměť RAM i ROM.

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'));
```

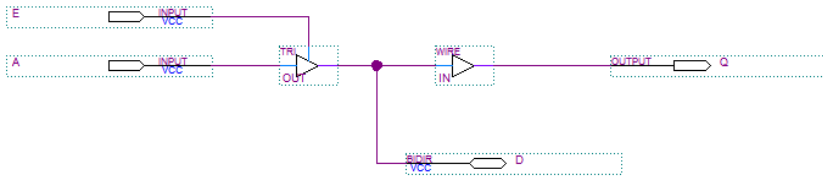
Nejprve deklarujeme typ ram_t, který představuje 256bytovou paměť RAM („pole osmibitových hodnot“), no a na druhém řádku vytvoříme jeho fyzickou reprezentaci – signál „ram“ bude mít hodně daleko do běžné představy „signálu“ coby datového vodiče, ale nebojte, za chvilku si ho „schováme“ do komponenty 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.

## 5.1 Obousměrná sběrnice

Ve VHDL můžeme relativně snadno implementovat obousměrnou třístavovou 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, prochází na tento signál hodnota ze vstupu 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;
```

## 5.2 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 4Kx1bit, 2Kx2, 1Kx4, 512×8, 512×9, 256×16, 256×18, 128×32 nebo 128×36 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 |
| 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.

*Poznámka „dvouportová" znamená, že nabízí dvě kompletní sady sběrnic, tj. 2x adresní, 2x datová a 2x řídicí sběrnice. Jedna část systému tak může do paměti třeba zapisovat, zatímco druhá část v tu samou dobu přes druhý port může číst.*

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

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

buované 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 paměti, 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 přistupujte 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;
```

## 5.3 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 pravděpodobně nejvhodnější varianta.

## 5.4 IP: Hotové paměti

Vývojové prostředí Quartus obsahuje knihovnu hotových komponent, uzpůsobených na míru konkrétním FPGA od Altery (Intellectual Properties, IP – později se k nim vrátíme). Ob-

dobnou 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, sčítačky) nebo právě paměti. U Altery můžeme tyto obvody využít tak, že je začleníme do architektury a vhodně nastavíme GENERIC MAP a PORT MAP.

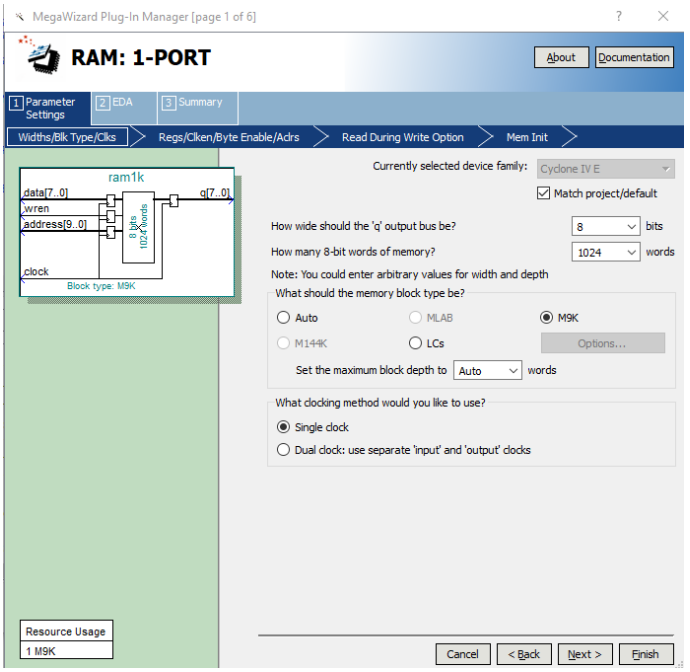
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

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;
```

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 *Megafunctions*. 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.

*V nových verzích IDE Quartus se tato možnost skrývá pod položkou IP Catalog. Po kliknutí se zdánlivě nic nestane, protože okno katalogu se otvírá v pravé části hlavního okna, vedle editoru.*




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

## 5.5 Pokus: Melodický zvonek

Kdysi jsem stvořil zvonek, který hrál melodii. Ne že by se takových nedaly koupit celé náruče za několik stokorun v nejbližším Horn-Baum-OBI, ale já chtěl takový, který hraje několikasekundovou smyčku reálné hudby, ne takový, který pípá pár tónů.

Nakonec jsem použil jednočip, v jehož FLASH byl nasamplovaný zvuk, a k němu jsem připojil D-A převodník, zesilovač a reproduktor.

S FPGA můžete udělat totéž. Místo D-A převodníku použijte třeba sigma-delta převodník z minulé kapitoly, použijte paměť ROM z této kapitoly a pak čítač, který načte aktuální hodnotu „samplu“ a pošle ji do převodníku.

Další vylepšení může být externí paměť FLASH – klidně sériová, která má výrazně vyšší kapacitu než interní paměť ROM. Připojení paměti přes rozhraní SPI není problém, o tom, jak ve VHDL stvořit SPI Master rozhraní, se ještě budeme bavit později.

# 6 Čítače

# 6 Čítače

## 6.1 Binární čítače

Svého času patřily čítače a paměti k maximu toho, co digitální integrované obvody dovolily – to bylo v dobách obvodů střední integrace, historicky označovaných jako MSI.

Nejjednodušší zapojení čítače je pomocí kaskády klopných obvodů typu T, tedy takových, které s každým celým pulsem, např. při každé vzestupné hraně, změní svou hodnotu. Fungují tedy jako dělička dvěma. Když takových obvodů za sebe zapojíte víc tak, že hodinový vstup prvního je zapojen na hodiny a u každého následujícího se hodinový signál tvoří součinem výstupů všech předchozích stupňů, získáte nejjednodušší čítač – binární s postupným přenosem.

*Binární čítač funguje i jako dělička kmitočtu – v prvním stupni ÷2, v druhém ÷4, ÷8, ÷16…*

Ve VHDL můžeme postupovat obdobně a nadefinovat několik děliček, ty pak pospojovat a vytvořit entitu „čítač“. Anebo to můžeme nechat na syntetizéru a předepsat, že čítač je čtyřbitová entita, která při každém hodinovém signálu zvýší svůj stav o 1.

```
ENTITY counter4B IS
    PORT (
        clk : IN std_logic;
        q : OUT std_logic_vector (3 DOWNTO 0)
    );
END ENTITY;

ARCHITECTURE main OF counter4B IS
    SIGNAL count : unsigned (3 DOWNTO 0);
BEGIN
    PROCESS (clk) BEGIN
        IF (rising_edge(clk)) THEN
                count <= count + 1;
        END IF;
    END PROCESS;
    q <= std_logic_vector(count);
END ARCHITECTURE;
```

Dobrý nápad je přidat k čítači vstup, kterým se hodnota čítače nuluje. Tento vstup může být

asynchronní, tj. nuluje obsah čítače kdykoli, nebo synchronní, který vynuluje čítač pouze při příchodu hodinového pulsu (obvykle při náběžné hraně). Nejprve si ukažme asynchronní verzi.

```
ENTITY counter4B IS
    PORT (
        clk : IN std_logic;
        reset : IN std_logic;
        q : OUT std_logic_vector (3 DOWNTO 0)
    );
END ENTITY;

ARCHITECTURE main OF counter4B IS
    SIGNAL count : unsigned (3 DOWNTO 0);
BEGIN
    PROCESS (clk, reset) BEGIN
        IF (reset = '1') THEN
            count <= (OTHERS => '0');
        ELSIF (rising_edge(clk)) THEN
            count <= count + 1;
        END IF;
    END PROCESS;
    q <= std_logic_vector(count);
END ARCHITECTURE;
```

Synchronní verze se liší v drobném detailu:

```
ENTITY counter4B IS
    PORT (
        clk : IN std_logic;
        reset : IN std_logic;
        q : OUT std_logic_vector (3 DOWNTO 0)
    );
END ENTITY;

ARCHITECTURE main OF counter4B IS
    SIGNAL count : unsigned (3 DOWNTO 0);
BEGIN
    PROCESS (clk) BEGIN
        IF (rising_edge(clk)) THEN
          IF (reset = '1')THEN
            count <= (OTHERS => '0');
```

```
          ELSE
            count <= count + 1;
          END IF;
        END IF;
    END PROCESS;
    q <= std_logic_vector(count);
END ARCHITECTURE;
```

Rozdíl je v tom, že u synchronní verze je proces citlivý pouze na hodinové pulsy, nikoli na změnu signálu RESET, a změna se kontroluje pouze po příchodu náběžné hrany. Dokud není detekována náběžná hrana, může se signál RESET měnit jakkoli, protože na jeho změny není proces citlivý.

Užitečná bývá i funkce LOAD – k čítači kromě výstupu přibyde stejně široký vstup a signál LOAD, kterým do čítače zapíšeme hodnotu na vstupu. Ukažme si takový čítač s asynchronním RESETem a synchronním LOADem:

```
ENTITY counter4B IS
    PORT (
        data : IN std_logic_vector (3 DOWNTO 0);
        load : IN std_logic;
        clk : IN std_logic;
        reset : IN std_logic;
        q : OUT std_logic_vector (3 DOWNTO 0)
    );
END ENTITY;

ARCHITECTURE main OF counter4B IS
    SIGNAL count : unsigned (3 DOWNTO 0);
BEGIN
    PROCESS (clk, reset) BEGIN
        IF (reset = '1') THEN
            count <= (OTHERS => '0');
        ELSIF (rising_edge(clk)) THEN
            IF (load = '1') THEN
                count <= unsigned(data);
            ELSE
                count <= count + 1;
            END IF;
        END IF;
    END PROCESS;
```

```
    q <= std_logic_vector(count);
END ARCHITECTURE;
```

Většina čítačů čítá směrem nahoru, tj v posloupnosti přirozených čísel 0, 1, 2, 3, … Ale pokud je potřeba, můžete snadno doplnit vstupy pro čítání dolů, resp. pro určení směru čítání.

## 6.2 Speciální čítače

Pokud jste pracovali s integrovanými obvody, možná si pamatujete na dvojici čítačů ze základní řady TTL 74xx, totiž obvody 7490 a 7493. Byly si velmi podobné, oba to byly čtyřbitové čítače (resp. jedno- a tříbitový) ale hlavní rozdíl mezi nimi byl ten, že 7493 počítal do 16, zatímco 7490 do deseti.

Dosahuje se toho (v reálných zapojeních) tím, že se do obvodu zapojí zpětná vazba na vstup RESET. Při dosažení určité hodnoty, třeba právě 10 (resp. binárně 1010) se čítač vynuluje, a tak reálně počítá od 0 do 9.

*Opravdu do důsledků vzato se na krátký okamžik na výstupech objeví hazardní stav 1010, než hradlo AND zpracuje, že jde o zakázaný stav, a než se projeví efekt RESETu.*

Přepis do VHDL je opravdu jednoduchý – přidáme jednu podmínku, a jak si s tím syntetizér poradí, to je už na něm.

```
ARCHITECTURE main OF counter4BCD IS
    SIGNAL count : unsigned (3 DOWNTO 0);
BEGIN
    PROCESS (clk, reset) BEGIN
        IF (reset = '1') THEN
            count <= (OTHERS => '0');
        ELSIF (rising_edge(clk)) THEN
            IF (load = '1') THEN
                count <= unsigned(data);
            ELSE
                IF (count = 9) THEN
                    count <= (OTHERS => '0');
                ELSE
                    count <= count + 1;
                END IF;
```

```
            END IF;
        END IF;
    END PROCESS;
    q <= std_logic_vector(count);
END ARCHITECTURE;
```

Občas se hodí čítače, které pracují s Grayovým kódem. Grayův kód je binární kód, jehož nejvýraznějším rysem je, že při přechodech mezi sousedními stavy (1 – 2 – 3 – 4 atd.) se mění pouze úroveň jednoho jediného bitu. Což je výhoda v případech, kdy se přenáší paralelně vícebitová hodnota a při změně více bitů „naráz" může vlivem odlišného zpoždění docházet ke vzniku přeslechů a falešných hodnot.

| **Hodnota** | **Binární** | **Grayův** |
|---|---|---|
| **0** | 00 | 00 |
| **1** | 01 | 01 |
| **2** | 10 | 11 |
| **3** | 11 | 10 |

Vidíte, že při přechodu ze stavu 1 do stavu 2 záleží na tom, aby se oba bity změnily naráz. Pokud by se měnily pomalu, mohou vzniknout mezistavy 00 (nižší bit se změní dřív) nebo 11 (vyšší bit se změní dřív).

Analogicky se přidává další bit, takže pro osm hodnot je posloupnost 000-001-011-010-110-111-101-100.

Grayův čítač se nejčastěji konstruuje jako binární čítač, za nímž je zapojen kombinační převodník z binárního kódu na Grayův. Je to obvodově velmi výhodné řešení, protože čítače jsou k dispozici již hotové a převodník jsou pouze hradla XOR.

Binární kód na Grayův převedeme tak, že:

- Nejvyšší bit ponecháme tak, jak je ($g_N$ <= $b_N$)
- Nižší bit XORujeme s hodnotou vyššího bitu ($g_{N-1}$ <= $b_N$ XOR $b_{N-1}$)

Opačný převod používá také hradla XOR, ale už nikoli paralelně, ale kaskádově, tj. musíte znát výslednou hodnotu vyššího bitu, abyste mohli spočítat předchozí.

- $b_N <= g_N$
- $b_{N-1} <= g_{N-1}$ XOR $b_N$

Další speciální čítače jsou například LFSR, tedy „posuvné registry s lineární zpětnou vazbou“, které lze používat pro generování náhodných čísel nebo k počítání polynomických výrazů. LFSR si ale zaslouží samostatnou kapitolu, tak se k nim ještě vrátíme.

## 6.3 Problém s přenosem

Vzpomínáte si na předvídání přenosu u velkých sčítaček? Jakmile šířka překročí určitou hranici, je zpoždění tohoto signálu už tak velké, že začne ovlivňovat výstupní hodnoty natolik, že je zkreslení měřitelné.

U čítačů nastává podobná situace. Protože každý další stupeň závisí na překlopení všech předchozích stupňů, zpoždění s každým dalším bitem šířky roste. U čtyřbitového čítače to není problém, ale třeba u čítače s šířkou 32 bitů je zpoždění citelné.

Syntetizéry proto využívají speciálních signálů uvnitř FPGA, zvaných „carry chain“, které spolu svážou jednotlivé logické buňky a zrychlí šíření těchto signálů. Na čipu jsou navíc vedeny nejefektivnějším možným způsobem, takže výsledné čítače mohou pracovat na frekvencích v řádech stovek MHz.

# 7 Automaty

# 7 Automaty

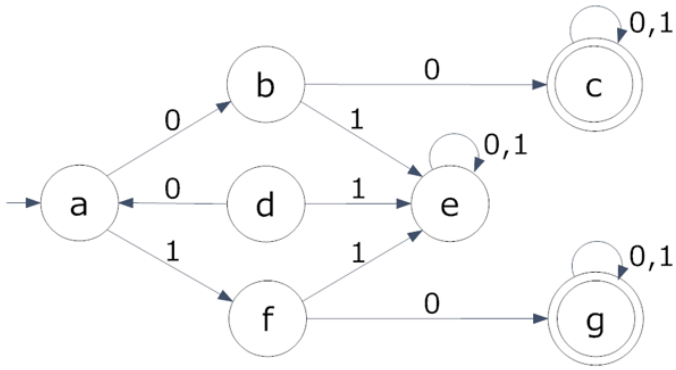
Trocha teorie o konečných stavových automatech, jejich implementaci ve VHDL, a jako bonus opravdové Hello World, tentokrát ne jako blikající LEDka, ale pěkně, řádně, přes sériové rozhraní!

## 7.1 Konečné automaty

Už jsme si říkali, že ve VHDL (mimo procesy) není, jako v programování, nějaké „dělej tohle a pak tamto“, ale 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 stavy 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 stavy 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 stavy, 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, stavy „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 správně 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.

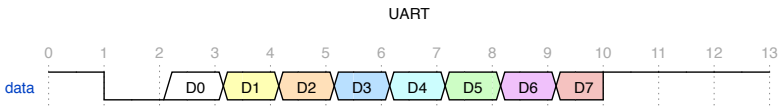
## 7.2 UART

Dobře, uznávám, je to přehnané. Nebudeme zatím 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 rychlost v bajtech jako „rychlost / 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 bitem, bez hardwarového řízení přenosu.




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 rychlost systémových hodin a vysílací rychlost 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. Později 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 50MHz / 9600 Hz = 5208,333… Použijeme hodnotu 5208, výsledný kmitočet tedy bude 9600,6144… 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 podělením 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 stavy: 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 zkopíruje do interního bufferu, vynuluje se výstup tx_ready, nastaví se počítadlo vysílaných 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 bity. 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 „synchronní náhradou“ signálu tx_send. Totiž – při testech jsem narážel 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 vysílacích 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;
                        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.

# 8 Hodinové domény

# 8 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 myslel.

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 vlastní zjednodušený úvod do celé problematiky.

## 8.1 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, předstihy 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 „Hello World“ blikač. Někdy je potřeba mít víc frekvencí, což byl případ sériového vysílače v minulé kapitole. Tam ale naštěstí šla nižší frekvence odvodit od té vyšší, prostým dělením. 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: *samplovací 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. Pokud 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 s hodinami?

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 impulsy 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 dosahuju fázového posunu) a opět rychlý clk3 s periodou 9ns (řádově stejně rychlý jako clk1, ale s drobným rozladěním, které vede k fázovým posunům).

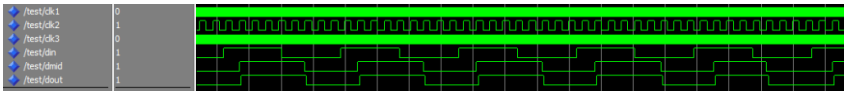

Na obrázku je v detailu vidět, jak se hodiny 1 a 3 posunou o 180° a jak hodiny 2 mají hranu úplně mimo.

Do série zapojuju 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.

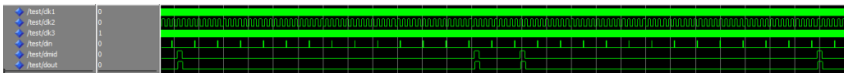
```
architecture bench of test is
signal clk1, clk2, clk3: std_logic:='0';
signal din, dmid, dout: std_logic:='0';

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
    strobesig: 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';
            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):




A takto signál strobe:



Myslím, že nebudu přehánět, když napíšu, že *máme u strobe vážný problém*… Prošel zhruba každý patnáctý. Pro dlouhé signály je tento přístup použitelný, pro krátké rozhodně ne.

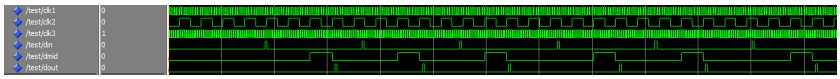
### Průchod krátkých pulsů

Pokud potřebujeme korektně zpracovat krátké pulsy, musíme použít složitější způsob. Uvnitř synchronizačního obvodu si udržujeme informaci o tom, že „přišel puls“ (samozřejmě řízenou

hodinovým kmitočtem zdroje), a pokud přišel, tak na výstupu vytvoříme puls 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:

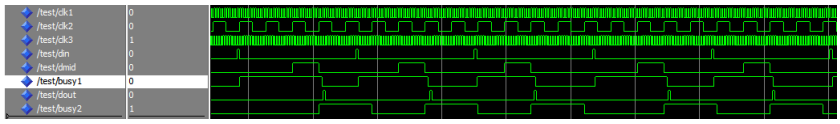


Vidíme, že pomalejší část poctivě tvoří pulsy podle krátkých pulsů na vstupu, s určitým zpožděním, a při konverzi z pomalých hodin na rychlé vznikají opět krátké pulsy.

Někdy může být výhodné, když dá převodník zdrojovému 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;
```




## Přenos 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 spoléhali 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, FIFO), kam jedna doména zapisuje a druhá čte.

## 8.2 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 zlíbí. 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 impulsy (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';
            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;
```

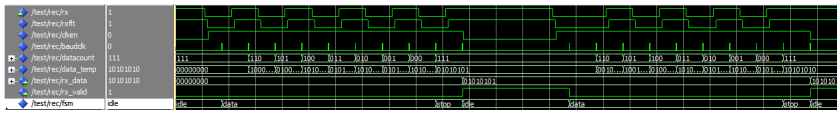
Přijímač posílá ven signál „rx_valid“, který oznamuje, že byla přečtena platná data. Nijak neřeší případné přetečení bufferu (taky neřeší, zda si už klient data vyzvedl), to je potřeba případně ošetřit v jiných částech.

## 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 tolik 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, vysílají se stejná data znovu. 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.

# 9 Generátor (pseudo)náhodných čísel

# 9 Generátor (pseudo)náhodných čísel

Elektronické obvody jsou, pokud jsou navrženy správně, deterministické. Nebo by měly být. To znamená, že pokud je v čase Tn 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 Tn+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 naprosto běžně dostupný. S šumem je to lepší. Principiálně vezmeme nějakou součástku, která generuje šum, například polovodičovou diodu, výstupní šum řádně 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 umožňuje, je dobré použít vstup od uživatele, například stisk tlačítek nebo pohyby myši, a využít toho, že intervaly stisknutí jsou řádově nižší než rychlost 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".

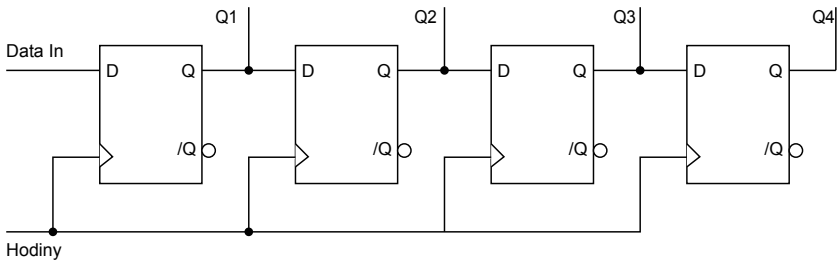
Využívá se techniky, kdy se sice čítají impulsy, ale hodnoty nejdou po sobě v očekávaném pořadí 1, 2, 3, 4, 5, 6…, ale např. u tříbitového čítače jako 1, 4, 6, 7, 3, 5, 2 (hodnota 0 se nevyskytuje). Takovou posloupnost odhalíme hladce, ale představte si, že použijeme čítač šestnáctibitový,

který má na výstupu čísla 1 až 65535 v pseudonáhodném pořadí. I tady se posloupnost objeví znovu, 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 $7{,}4\times10^{42}$ sekundách, cože je $2.37\times10^{35}$ let. Chcete-li to převést na biliony, je to $2.37\times10^{23}$ 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.

## 9.1 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 |
| **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 |
| **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 |
| **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 (2N-1 stavů) vystačíme s dvou-, či čtyřvstupovými hradly. I LFSR o délce 168 bitů postavíme jednoduše, zavedením zpětné vazby signálem (Q151 xor Q153 xor Q166 xor Q168 – číslujeme od 1). Vhodné koeficienty naleznete 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 najdete vhodné kombinace čísel. Pro čítače dlouhé 2 až 786 bitů je najdete na adrese https://datacipy.cz/lfsr_table.pdf. 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 dostatečně 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 pří-

padě 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;
```

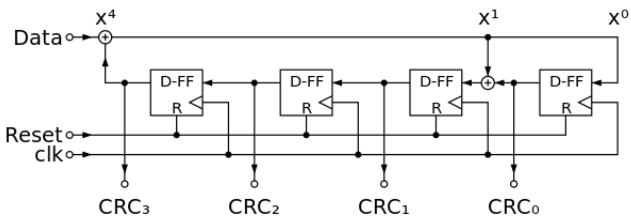
Vidíme posuvný registr o šířce 31 bitů (čísluju nikoli od nuly, ale od jedničky). Zápis je dostatečně obecný, takže při úpravě na jinou šířku registru stačí změnit pouze zpětnovazební funkci. Pro zajímavost ještě architektura one-to-many:

```
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).



# 10 IP, OpenCores a hardware s FPGA

# 10 IP, OpenCores a hardware s FPGA

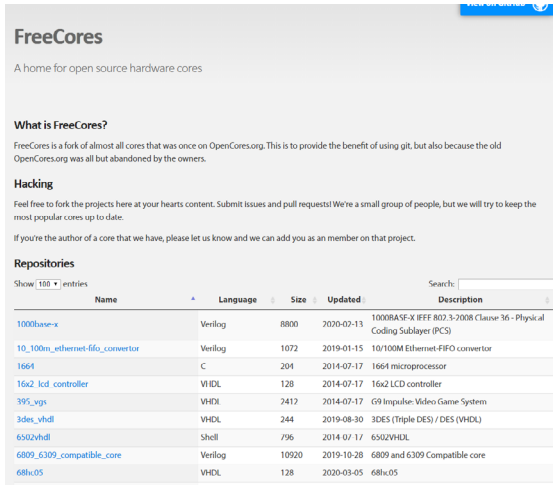
Asi největší radost jsem při učení se VHDL zažil ve chvíli, kdy jsem objevil OpenCores.

Všichni výrobci dodávají takzvané IP – předpřipravené moduly s danou funkčností, například paměti, interface nebo procesory, které můžete použít ve svých konstrukcích. Některé bezplatně, jiné po zaplacení licenčních poplatků. Zkratka IP totiž v tomto kontextu znamená „Intellectual Property“, čili *duševní vlastnictví*. Bývá pravidlem, že takový IP modul od výrobce je „black box“ s definovaným chováním a definovanými porty, ale jeho konstrukce je uzamknutá.

Vznikla ale open-source iniciativa OpenCores (https://opencores.org/), která se snaží přinášet základní moduly v podobě otevřených kódů. Projekt pochází ještě z 90. let, a je to na něm, bohužel, leckde znát. Například při správě zdrojových souborů pomocí SVN. Navíc vypadá, že se o něj delší dobu už nikdo nestará. Což je škoda, protože na OpenCores vznikla spousta zajímavých věcí, jako například koncept mezimodulové sběrnice WishBone.

Naštěstí vznikl klon pod názvem FreeCores, spravovaný a organizovaný pomocí GitHubu (https://freecores.github.io/). Většina „živých“ projektů stejně přešla na GitHub, a tak je logické, že tamtéž vznikl i rozcestník.

Použití modulů z OpenCores / FreeCores je jednoduché: vyberete si, jaké moduly do svojí konstrukce potřebujete, stáhnete si je (buď pomocí Gitu, nebo jako .ZIP), rozbalíte, a uvnitř většinou naleznete knihovny (ve VHDL nebo Verilogu), někdy i testbench a dokumentaci. Knihovny naimportujete do svého projektu, a na požadovaném místě už jen použijete potřebnou komponentu.



FreeCores

A home for open source hardware cores

**What is FreeCores?**

FreeCores is a fork of almost all cores that was once on OpenCores.org. This is to provide the benefit of using git, but also because the old OpenCores.org was all but abandoned by the owners.

**Hacking**

Feel free to fork the projects here at your hearts content. Submit issues and pull requests! We're a small group of people, but we will try to keep the most popular cores up to date.

If you're the author of a core that we have, please let us know and we can add you as an member on that project.

**Repositories**

Show 100 entries Search:

| Name | Language | Size | Updated | Description |
|---|---|---|---|---|
| 1000base-x | Verilog | 8800 | 2020-02-13 | 1000BASE-X IEEE 802.3-2008 Clause 36 - Physical Coding Sublayer (PCS) |
| 10_100m_ethernet-fifo_convertor | Verilog | 1072 | 2019-01-15 | 10/100M Ethernet-FIFO convertor |
| 1664 | C | 204 | 2014-07-17 | 1664 microprocessor |
| 16x2_lcd_controller | VHDL | 128 | 2014-07-17 | 16x2 LCD controller |
| 395_vgs | VHDL | 2412 | 2014-07-17 | G9 Impulse Video Game System |
| 3des_vhdl | VHDL | 244 | 2019-08-30 | 3DES (Triple DES) / DES (VHDL) |
| 6502vhdl | Shell | 796 | 2014-07-17 | 6502VHDL |
| 6809_6309_compatible_core | Verilog | 10920 | 2019-10-28 | 6809 and 6309 Compatible core |
| 68hc05 | VHDL | 128 | 2020-03-05 | 68hc05 |

Chcete vlastní počítač? Můžete začít výběrem jeho srdce: Pro procesor Z80 jsou tu asi čtyři implementace (nepočítám Z8, eZ8 ani Z180), 8080 máte třikrát, 6502 rovnou pětkrát, 6809 najdete taky, máte tu od Intelu 8051, 4004, Motorolu 68hc05 i 68hc08, dokonce i 68000, procesory s architekturami AVR, ARM i MIPS, procesory PDP-8 i PDP-11, různé RISCové stroje, miniaturní jádra i VLIW...

Paměť? Možná se vám bude hodit některý z připravených řadičů SDRAM, rozhraní pro CompactFlash i pro SPI Flash, nebo pro SD karty.

S uživatelem můžete komunikovat přes monitor. Vyberte si, nabídka zahrnuje nejrůznější displeje s výstupy PAL, NTSC, VGA nebo HDMI. I řadiče pro LED moduly tu jsou. Pro fajnšmekry pak dekodéry i enkodéry (M)JPEG, dekodér H.264/AVC, MPEG2, nebo i implementace ZX Spectrum ULA.

Když už jsme nakousli generování videa – nejsou to jediné aritmetické operace, realizované pomocí hardwaru. Máte tu k dispozici různé generátory CRC, modul pro AES128, FFT, DCT, aritmeticko-logické jednotky pro práci s reálnými čísly, komponenta pro kompresi LZRW1 i různé matematické koprocesory.

V sekci Crypto najdete jak různé AES / 3DES / RSA / Twofish / XTEA komponenty, tak i počítače hashů MDx, SHAx, a dokonce i specializované bitcoinové komponenty pro vlastní „BTC minery“.

DSP zase nabízí součásti pro digitální zpracování signálů, především nejrůznější filtry a procesory.

Pro komunikaci se světem zase využijete nejrůznější řadiče Ethernetu (10/100M, 1G, 10G), někdy i včetně UDP / IP stacku. Samozřejmostí jsou různé U(S)ART obvody, rozhraní PS/2 nebo řadiče sběrnic CAN, RS-245, SPI, I2C i USB. Pokud je vaším cílem vytvořit například přídavnou kartu pro PC, přijde vhod řadič PCI / PCIe. Pro připojení disků užijete implementaci řadiče SATA, u malých zařízení zase rozhraní pro SD/MMC/SDHC karty. Nadšence do zvuku potěší implementace rozhraní SPDIF.

Výčet můžeme uzavřít open-source implementací zvukových čipů AY (Yamaha) a OPL3, hardwarovou obsluhou filesystému FAT, kodeku pro Ogg Vorbis, přijímačem FM signálu nebo detektorem pohybu v obrazu.

OpenCores / FreeCores vám mohou ušetřit spoustu času a urychlit vývoj vašeho zařízení snů. Mohou se stát ale také zdrojem nočních můr: mnohé projekty byly už před lety opuštěné, leckdy v polododělaném stavu, kde některé funkce nejsou implementované atd. Ale to je riziko světa open-source.

## 10.1 Multicomp

Grant Searle (http://searle.wales), známý tvůrce jednoduchých počítačových konstrukcí, představil svou počítačovou skládačku Multicomp. Používá kit, který jsem vám doporučoval – ten nejmenší s FPGA Cyclone II a označením EP2C5 (kompletní typ je EP2C5T144C8N). V tomto kitu si můžete vybrat z následujících komponent:

- CPU:
  - Z80
  - 6502
  - 6809
- Rozhraní
  - Černobílé video a klávesnice PS/2
  - Barevné video (TV / VGA) a klávesnice PS/2
  - Terminál (RS-232)
- Paměť
  - Interní 1, 2 nebo 4 kB RAM
  - Externí až do 64 kB RAM (použijte paměť, která pracuje při 3.3 V!)
- ROM
  - 8 kB s interpretem jazyka BASIC

Můžete je nakombinovat téměř libovolně, s jednou výjimkou: některé terminály, zejména barevné s velkým rozlišením, jsou tak náročné na paměť, že už nezbývá žádná interní pro systém a musíte vždy připojit externí. Pokud nechcete připojovat vůbec žádné externí obvody, jen sériový terminál, zvolte třeba Z80 + 4 kB RAM. Pokud připojíte monitor a PS/2, můžete použít třeba procesor 6502, 2 kB RAM a textové monochromatické rozhraní 80x25, případně použít barevné v rozlišení 40x25.

Když ke kitu připojíte 64 kB SRAM (pravděpodobně asi jeden 128k čip, například AS6C1008,

v pouzdru DIL, s tím, že polovina nebude použitá), můžete si zkusit i plnou výbavu, třeba s procesorem Z80, barevným displejem 80x25 a SD kartou. V takovém případě můžete s minimální úpravou svůj stroj předělat tak, aby fungoval s operačním systémem CP/M–i takovou úpravu autor nabízí na svém webu:

http://searle.x10host.com/Multicomp/cpm/fpgaCPM.html

Když si stáhnete z autorova webu balík s potřebnými kódy, zjistíte, že jste získali několik různých modulů, které si musíte podle autorova návodu a v závislosti na tom, jakou konfiguraci jste zvolili, v hlavním modulu správně pospojovat.

V adresáři Microcomputer je hlavní komponenta (microcomputer. vhd), v níž proběhnou veškeré úpravy. Kromě ní jsou zde připravené tři paměti RAM (1, 2 a 4 kB).

V adresáři ROMS jsou autorem upravené a připravené paměti ROM s třemi různými verzemi interpreteru jazyka BASIC, pro každý použitý procesor jedna.

A konečně v adresáři Components jsou jednotlivé procesory a další komponenty. Procesory definují komponenty T65 (procesor 6502), CPU09 (procesor 6809) a T80 (procesor Z80) – všechny najdete v OpenCores. Kromě nich jsou zde i komponenty pro práci se SD kartami, pro UART a pro terminál (grafické rozhraní a klávesnici).

Multicomp se dočkal implementace pro různé vývojové kity:

https://github.com/douggilliland/MultiComp

Neal Crook rozvinul konfiguraci s procesorem 6809 a doplnil různé vybavení pro tento procesor (implementaci CamelForth, FLEX, CUBIX, NITROS9, FUZIX a emulátor exec09):

https://github.com/nealcrook/multicomp6809

## 10.2 MiST

Jakmile ceny FPGA klesly na přijatelnou úroveň, objevila se spousta konstruktérů, kteří s těmito obvody začali stavět nejen nové konstrukce, ale i „staro-nové“ konstrukce. Jedním z nich je MiST – deska, kterou navrhl Till Harbaum. Jeho cílem bylo mít desku s obvodem FPGA, v němž by mohl re-implementovat (nikoli „emulovat“, ale „znovu-vytvořit“) počítače Amiga a Atari ST. Proto MiST – aMIga a ST.

V době psaní této knihy nabízí několik prodejců desku MiST 1.4 za ceny okolo 200 EUR.

https://github.com/mist-devel

Deska je postavena na výkonném FPGA Cyclone III typu EP3C25, k němuž je připojeno 32 MB SDRAM (16bitová sběrnice). O periferie se stará mikrokontrolér ARM AT91SAM7S56, rozhraní USB zařizují obvody MAX3421E a TUSB2046.

Deska nabízí 4 USB porty, výstup na VGA (3x6 bitů barevné hloubky), stereo audio výstup, MikroUSB pro napájení a nahrávání software, 2 konektory pro klasické joysticky (CANON-9), slot pro SD kartu, LED a tlačítka.

Původní záměr se autorovi takříkajíc vymknul z rukou, takže dnes nabízí MiST kromě strojů Amiga (500 / 600 / 1200) a Atari ST / STE i spoustu dalších strojů, implementovaných ve VHDL či Verilogu.

Kromě ZX Spectra (s AY, DivMMC a ESXDOS), ZX81, Sam Coupe, Amstradu CPC, BBC Micro, Apple II a Macintoshe, Archimeda, MSX, Oric, či legend C64 a Atari 800 / 130 existují i konfigurace pro herní konzole Atari 2600, SEGA Genesis, Coleco, (S)NES a pro velké množství arkádových hracích automatů.

Na stránkách projektu MiST je obrovské množství nejen hotových a přeložených konfigurací pro ty, co si chtějí jen zkusit pracovat se starým strojem, ale také veškeré zdrojové kódy k jednotlivým strojům.

Při pohledu na zdrojové kódy například ZX Spectra si uvědomíte, jak jednoduché to vlastně celé bylo. Procesor Z80 (implementace OpenCores T80), k tomu trocha logiky, reimplementovaná ULA, a víc není potřeba.

## 10.3 ZX Spectrum Next

Možná jste sledovali na Kickstarteru kampaň na vznik tohoto retropočítače. Jde vlastně o znovuvytvoření ZX Spectra, ale tentokrát pro 21. století. Podobné pokusy tady byly, například Speccy 2010, autoři Spectra Next to ale vzali od podlahy včetně vlastního plastového pouzdra (které, jak se zdá, zapříčinilo několikaletý skluz v dodávkách).

Základem Spectra NEXT je Xilinx Spartan 6 (XC6SLX16), v němž autoři udělali staré dobré ZX Spectrum, jen v každém detailu o něco pokročilejší: přidali vlastní instrukce do jádra Z80, vylepšili ULA, zvětšili paměť, přidali DMA, AY je tu rovnou třikrát, můžete připojit modul ESP8266 a komunikovat ze Spectra po WiFi, ...

Bohužel zdrojové kódy nejsou v době psaní této knihy k dispozici, ale i tak zde tento stroj zmíním.

## 10.4 Gameduino

Před mnoha lety vzniklo Gameduino, přídavná deska (shield) k Arduinu, která přidávala VGA grafiku a zvuk, takže Arduino mohlo sloužit jako jednoduchý herní stroj.

https://www.excamera.com/sphinx/gameduino/

Samozřejmě bystřejší tuší, že kouzlo bylo opět v čipu FPGA, tentokrát Xilinx Spartan XC3S200. Kromě tohoto obvodu byly na desce už jen převodníky mezi 5 V a 3.3 V logikou, konektor VGA a konektor pro zvuk.

Gameduino se připojovalo jako běžná periferie po sběrnici SPI a tvářilo se jako 32 kB paměti. Kromě možností, které byste od něj očekávali, tedy zobrazování obrazu, přehrávání zvuku a nějaké základní práce se sprity, se v něm skrývalo i jedno překvapení: mikroprocesor J1, který bylo možné programovat v jazyce FORTH. Tento procesor měl přímý přístup k video paměti a vy jste jej mohli použít jako svého druhu grafický koprocesor.

https://www.excamera.com/sphinx/gameduino/coprocessor.html

Tento koprocesor má interní 16bitovou sběrnici, násobičku 16x16 bitů, barrel shifter, hardwarové zásobníky a výkon okolo 50 MIPS. Pokud jste o tomto procesoru nikdy neslyšeli, není divu. Navrhnul si ho autor Gameduina pro své vlastní potřeby. A pomalu se i my blížíme do okamžiku, kdy si navrhneme vlastní mikroprocesor.

*A dokonce si na procesoru J1 ukážeme, jak se popisují obvody ve Verilogu, ale to až na konci knihy.*

Naštěstí je celé Gameduino pod licencí BSD / GPL, takže jsou k dispozici zdrojové kódy volně k použití i pro vlastní inspiraci.

# 11 OMEN Alpha, tentokrát ve FPGA

## 11 OMEN Alpha, tentokrát ve FPGA

V knize *Porty, bajty, osmibity* jsem popisoval osmibitový počítač s procesorem 8085, nazvaný OMEN Alpha. Kromě procesoru obsahoval 32 kB RAM, 32 kB EEPROM, sériové rozhraní 68B50 a paralelní rozhraní 8255. S drobnými omezeními (danými především kapacitou paměti) můžeme tuto konstrukci převést do FPGA, a využijeme k tomu právě moduly, dostupné jako open-source.

Hned na začátku si ale úlohu zjednodušíme: místo procesoru 8085, jehož otevřená implementace není v době psaní textu dostupná (respektive nenašel jsem ji, *pozn. aut.*), použijeme procesor 8080, a to dobře dokumentovanou a ověřenou implementaci light8080, dostupnou pod licencí GNU LGPL:

https://github.com/jaruiz/light8080

Jedná se o mikroprocesorové jádro řízené mikroprogramově (později si ukážeme principy mikroprogramovaných procesorů). My jej teď použijeme jako hotovou komponentu a nebudeme se zajímat o to, jak je uvnitř vyřešený.

*Našel jsem i implementaci 8085 s názvem STD8085, ale je dostupná jako licencovaná komponenta (IP). K výukovým a testovacím účelům můžete použít její demoverzi STD8085_DEMO, která je omezena na 512 byte paměti (adresová sběrnice má jen 9 bitů).*

O paměti a tom, jak ji vytvořit, už řeč byla. Zbývá tak jen sériové rozhraní. Můžeme použít implementaci *acia6850*, která je dostupná pod licencí GNU GPL v rámci OpenCores projektu System09, můžeme použít (s Grantovým svolením) UART kompatibilní s 6850 z Multicompu, anebo klidně i náš vlastní UART.

Paralelní port přijde na řadu později, ten není nezbytně nutný.

*Zdrojové kódy jsou ke stažení na webu GitHub:*
*https://github.com/datacipy/VHDL*

Začneme vytvořením projektu. Pojmenujeme ho alpha, vybereme typ FPGA Cyclone II EP2C5T144C8 a zatím nebudeme nic přidávat, jen doklikáme na konec.

Pokračujeme vytvořením pamětí. Použijeme k tomu už zmiňovaný MegaWizard plug-in Manager. Z menu „Memory compiler“ vyberte „RAM:1-PORT“, jako název zadejme

„ram4k.vhd“, v dalším kroku určíme šířku 8 bitů a velikost 4096 bajtů, nezapomeňte zadat uložení v blocích M4K, a pak už všemu můžeme nechat defaultní hodnoty.

Znovu v MegaWizardu vybereme paměť, tentokrát „ROM: 1-PORT“, název bude „rom4k.vhd“, šířka 8 bitů, velikost 4096 bajtů, a v kroku, kde se určuje obsah paměti, vyberte HEX soubor s monitorem. *Ano, obě paměti mají velikost jen 4 kB; větší se do našeho FPGA nevejdou.*

Stáhněte si knihovny ACIA6850 a light8080 a rozbalte potřebné VHDL soubory.

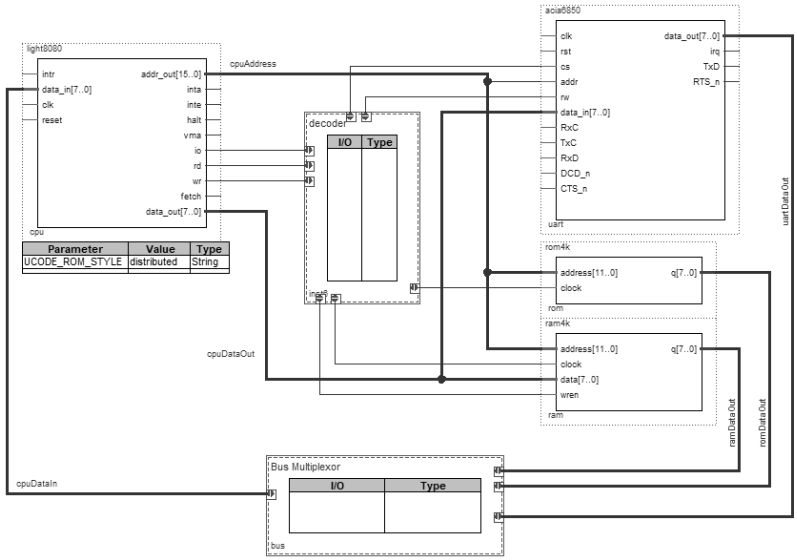
*Obě knihovny najdete ve zdrojových kódech.*

Vytvořte si prázdný soubor alpha.vhd a přidejte ho do projektu. Přidejte také ram4k.vhd, rom4k.vhd, z knihovny ACIA6850 přidejte acia6850.vhd a aciaclock.vhd, a konečně z knihovny light8080 přidejte light8080.vhdl a light8080_ucode_pkg.vhdl.

Teď máme vše potřebné pohromadě a můžeme začít zapojovat. Všimněte si, že jak procesor, tak paměti i ACIA mají dvě datové sběrnice, vstupní a výstupní. Díky tomu můžeme bez problémů napřímo připojit výstupní datovou sběrnici procesoru na vstupní sběrnice periferií (ty jsou řízeny vlastními povolovacími vstupy). Výstupy periferií ale musíme do procesoru přivádět v závislosti na tom, která periferie je právě adresovaná. Budeme tedy potřebovat nějaký přepínač (bus isolator).

Zároveň budeme potřebovat blok s kombinačními logickými obvody, který ze signálů rd, wr, io vytvoří signály pro čtení z paměti, čtení z I/O, zápis do paměti a zápis do IO. Také pro generování signálů ramcs, romcs, uartcs (výběr periferního obvodu), ramwr a uartwr (aktivní, když se zapisuje do těchto obvodů).

Hrubý „ideový“ návrh celého zapojení vidíte na obrázku:




S vnějškem bude celá komponenta „alpha“ propojena třemi signály: clk, TxD a RxD (pro sériové rozhraní).

```
ENTITY alpha IS
    PORT (
        clk : IN std_logic;
        RxD : IN std_logic; -- Vstup
        TxD : OUT std_logic -- Vystup
    );
END;
```

Začneme *instanciací entit* (zapíšeme, které použijeme a jak jsou propojené):

```
-- cpu
cpu : ENTITY work.light8080 PORT MAP (
    rd => rd, wr => wr,
    clk => cpuClock,
    data_out => cpuDataOut,
    data_in => cpuDataIn,
    addr_out => cpuAddress,
    io => iom,
```

```
    intr => '0', --bez preruseni
    reset => reset
    );

-- ROM
rom : ENTITY work.rom4k PORT MAP (
    address => cpuAddress(11 DOWNTO 0),
    clock => clk,
    q => romDataOut
    );

-- RAM
ram : ENTITY work.ram4k PORT MAP (
    address => cpuAddress(11 DOWNTO 0),
    clock => clk,
    data => cpuDataOut,
    wren => ramwr,
    q => ramDataOut
    );

-- UART
uart : ENTITY work.acia6850 PORT MAP (
    clk => cpuClock,
    rst => reset,
    cs => uartcs,
    addr => cpuAddress(0),
    rw => uartwr,
    data_in => cpuDataOut,
    data_out => uartDataOut,
    RxC => uartClock,
    TxC => uartClock,
    RxD => RxD,
    TxD => TxD,
    DCD_n => '0',
    CTS_n => '0'
    );
-- baud
baud : ENTITY work.aciaClock GENERIC MAP (50_000_000, 115_200) PORT MAP(clk, uartClock);
```

Všechny jsou nadefinované a mají přidělené signály. Měli bychom si je rovnou nadeklarovat:

```
-- cpu
SIGNAL wr : std_logic;
SIGNAL rd : std_logic;
SIGNAL iom : std_logic;
SIGNAL cpuAddress : std_logic_vector(15 DOWNTO 0);
SIGNAL cpuDataOut : std_logic_vector(7 DOWNTO 0);
SIGNAL cpuDataIn : std_logic_vector(7 DOWNTO 0);

SIGNAL cpuClock : std_logic;
SIGNAL reset : std_logic := '1';

--bus logic
SIGNAL memwr, memrd : std_logic;SIGNAL iowr, iord : std_logic;
SIGNAL ramwr : std_logic;
SIGNAL ramcs, romcs : std_logic;
SIGNAL uartcs : std_logic;
SIGNAL uartwr : std_logic;

--rom
SIGNAL romDataOut : std_logic_vector(7 DOWNTO 0);

--ram
SIGNAL ramDataOut : std_logic_vector(7 DOWNTO 0);

--uart
SIGNAL uartDataOut : std_logic_vector(7 DOWNTO 0);
SIGNAL uartClock : std_logic;
```

Teď potřebujeme vygenerovat signály memrd, memwr, iord a iowr:

```
memrd <= rd AND NOT iom;
memwr <= wr AND NOT iom;
iord <= rd AND iom;
iowr <= wr AND iom;
```

Následují signály pro výběr periferií. Namapujeme paměti tak, aby ROM začínala od 0h (0000h – 0FFFh), RAM bude na konci prostoru (F000h – FFFFh) a UART na adresách DEh, DFh:

```
ramcs <= '1' WHEN cpuAddress(15 DOWNTO 12) = "1111"
    ELSE '0';
romcs <= '1' WHEN cpuAddress(15 DOWNTO 12) = "0000"
```

```
    ELSE '0';
uartcs <= '1' WHEN cpuAddress(7 DOWNTO 1) = "1101111"
    ELSE '0';

ramwr <= memwr AND ramcs;
uartwr <= NOT (iowr AND uartcs);
```

Použijeme děličku kmitočtu *aciaClock* pro vygenerování procesorového hodinového signálu:

```
cpuClk : ENTITY work.aciaClock
           GENERIC MAP (4, 1)
           PORT MAP(clk, cpuClock);
```

Pokud bychom kmitočet alespoň trochu nesnížili, nestačily by použité synchronní paměti vydat data v okamžiku, kdy je procesor očekává, a systém by neběžel.

Vůbec časování může připravit řadu nečekaných problémů, hlavně v případě, že používáte komponenty, u nichž není zcela jasné, kdy přebírají jaká data. Proto opakuju: **testujte, testujte, testujte, modelujte, zkoušejte!** Jednou dobře odzkoušené zapojení vydá za tucet nadávek na mizernou dokumentaci.

Nakonec přidáme (přesně z důvodu, zmíněného v předchozím odstavci) přepínání datové sběrnice, vytvořené jako proces, synchronizovaný s procesorovými hodinami. *Důvod je ten, že light8080 v jednom cyklu vystaví adresu a signál memory read, ale data přečte s další vzestupnou hranou, kdy zároveň adresu zneplatní a memory read jde zpět do 0. A rychlé FPGA stihnou mezitím přepnout vstupní datovou sběrnici. U jiných procesorů, třeba u T80, což je VHDL implementace Zilogu Z80, se takové harakiri vůbec nemusí dělat,*

```
    PROCESS (cpuClock)
    BEGIN
        IF rising_edge(cpuClock) THEN
            -- sbernice
            IF (romcs = '1' AND memrd = '1') THEN
                cpuDataIn <= romDataOut;
            ELSIF (ramcs = '1' AND memrd = '1') THEN
                cpuDataIn <= ramDataOut;
            ELSIF (uartcs = '1' AND iord = '1') THEN
                cpuDataIn <= uartDataOut;
            ELSE
                cpuDataIn <= x"00";
            END IF;
```

```
        END IF;
    END PROCESS;
```

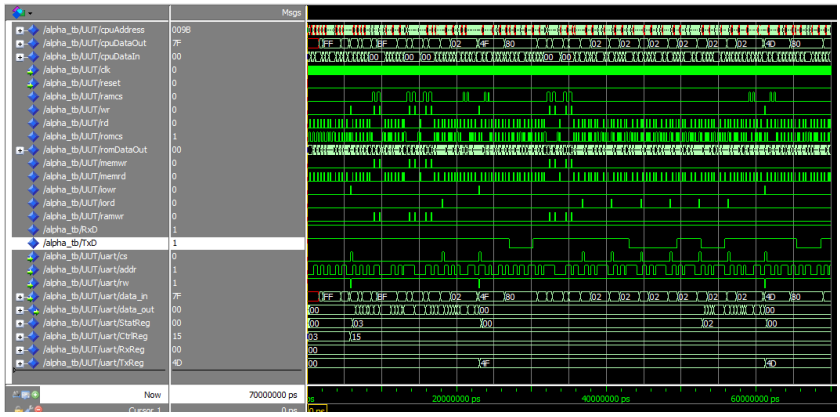
No a poslední signál je „automatický RESET“ – jednoduchá procedura, která ponechá signál RESET po spuštění pět cyklů hodin v úrovni „1“, a poté jej shodí do úrovně „0“:

```
PROCESS (cpuClock)
    VARIABLE resetDuration : INTEGER := 5;
BEGIN
    IF rising_edge(cpuClock) THEN
        IF resetDuration = 0 THEN
            reset <= '0';
        ELSE
            resetDuration := resetDuration - 1;
            reset <= '1';
        END IF;
    END IF;
END PROCESS;
```

A to je vše. Tedy co se týče psaní.

Teď stačí jen správně nastavit tři signály, které Alpha vystavuje, a přiřadit je podle zapojení kitu, tj. hodinový vstup na pin 17 a piny TxD, RxD na volné a dostupné piny, k nim připojit USB převodník (nezapomeňte na úroveň 3,3V!) a vyzkoušet zapojení naostro.

Ale i bez toho si můžete vše vyzkoušet alespoň v emulátoru. Nastavte si ale děličku pro UART třeba na 25M, ať nečekáte věčnost na nějaký projev života na výstupu TxD. Na obrázku je dobře vidět, jak se poslala hodnota 4Fh (nulový start bit, pak čtyři jedničkové, dva nulové, jeden jedničkový, jeden nulový, a jedničkový stop bit):




Při vlastních experimentech můžete vyzkoušet různé mikroprocesory, různé konfigurace a vylepšení, i když omezení dostupné paměti může být citelné.

Jeden z prvních experimentů, který jsem ve VHDL napsal, byl mikropočítač PMI-80. Tento jednodeskový počítač měl 1 kB RAM a 1 kB ROM, klávesnici 5x5 tlačítek a sedmisegmentový displej. Taková konfigurace se vejde i do toho nejmenšího kitu s Cyclone II.

Teoreticky by se mohla do stejného kitu vejít i implementace legendárního československého mikropočítače JPR-1 s 1 kB RAM. Ověřená je implementace mikropočítače ZX-81 – původní s názvem ZX97, pozdější s názvem ZX01. Zkrátka u prvních domácích osmibitů asi nenajdete takový, který by vzdoroval přepsání do FPGA.

V příkladech ke knize (https://datacipy.cz/) najdete například implementaci OMEN Alpha s procesorem Z80, který využívá integrovaný čip SDRAM, nabízí možnost přepínání bank pamětí, implementuje přístup k periferiím na kitu, používá klávesnici PS/2 a VGA monitor jako výstup atd.

# 12 Generování VGA videosignálu

# 12 Generování VGA videosignálu

Víte, co v mém 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!

Bezpočtukrát 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 piplačky. 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 tutoriálem...

FPGA totiž často obsahují obvody, zvaný PLL (česky *fázový závěs*), které umožňují vstupní kmitočet vynásobit i vydělit celočíselnou hodnotou, a tak 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 (1600×1200). 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.

## 12.1 VGA teoreticky

Videosignál se skládá ze snímků (u prokládaného videa je to komplikovanější o půlsnímky, ale prokládané video snad už nikdo nepoužívá). 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 (Hz).

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 mikrosekund), ale u VGA je praktičtější udávat je tak, jak jsem napsal, tedy v pixelech a řádcích.

Další důležitý údaj je „pixelová frekvence“. Tedy rychlost, jakou je potřeba posílat pixely do monitoru. Tato rychlost 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 = 20ns. To je naše základní jednotka. Jeden řádek tedy zabere 800 pixelů (800 * 20 = 16 μs) plus neviditelnou část: 56 pixelů front porch, 120 pixelů horizontální synchronizace, 64 pixelů back porch. Tedy 1040 pulsů hlavních hodin = 20,8 μs.

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 μs, tak vynásobením dostáváme trvání jednoho snímku 13,8528 ms, což dává opravdu obnovovací frekvenci 72 Hz (přesně 72,187572 periodicky).

## 12.2 Synchronizace

Na jednu stranu jsou monitory poměrně tolerantní, odpustí vám drobné (ale jen opravdu drobné) rozladění frekvencí a dokáží se správně zasynchronizovat. 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 nejlépe 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. U LCD panelů žádný paprsek není a jeho návrat tedy neprobíhá. Ale princip zůstal. Proto 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 v klidu 1 a puls 0, nebo jestli je v klidu 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.

## 12.3 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 si koupil kit „OMDAZZ", který jsem zmiňoval na začátku. Má zabudovaný konektor VGA, a má ho připojený tak, že každou barvu tvoříte jedním bitem. Žádná hitparáda to není, ale jako základ stačí.

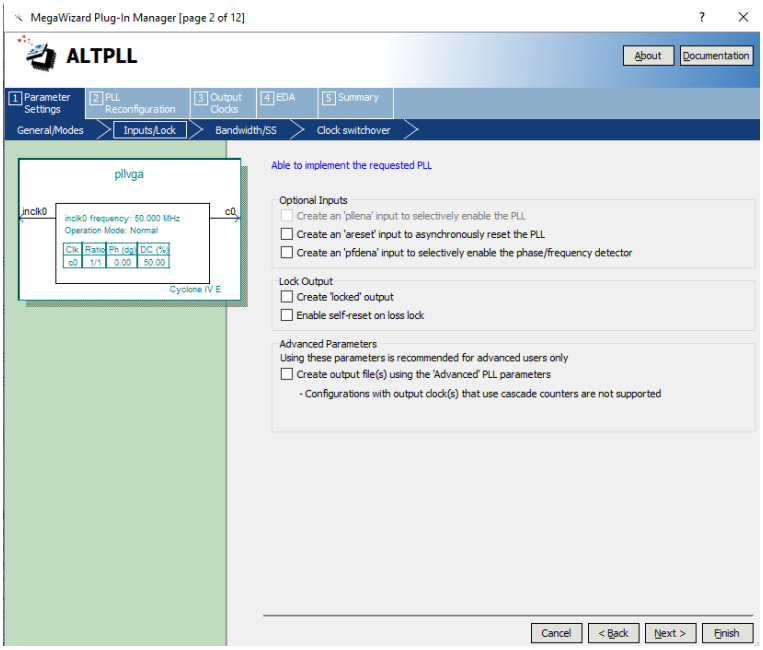
## 12.4 PLL

Pro generování obrazu ještě potřebujeme ten známý „pixelový kmitočet". K jeho vygenerování slouží právě PLL – fázový závěs (Phase Locked Loop). Použití PLL se liší u jednotlivých výrobců FPGA. Já se podržím toho, co nabízí Altera.

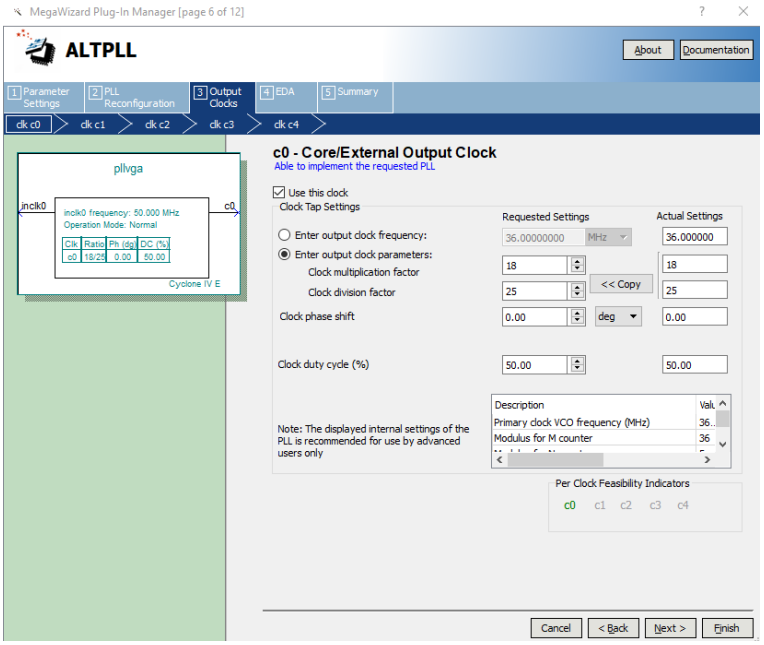
Nejprve vyberte z menu Tools možnost „MegaWizard Plug-In manager". Nechte „vytvořit novou megafunkci" a v dalším kroku vyberte „altpll" (je v oddíle I/O). Jako jazyk vyberte VHDL, vyberte vhodné umístění a jméno souboru („pll" není špatné) a pokračujte dál v průvodci.

*V nových verzích IDE Quartus se tato možnost skrývá pod položkou IP Catalog. Po kliknutí se zdánlivě nic nestane, protože okno katalogu se otvírá v pravé části hlavního okna, vedle editoru.*

Na straně 3 vyberte rychlost vašeho FPGA (udává to číslice na konci označení, já mám EP4CE6E22C8N – a rychlost je právě těch 8). Zadejte vstupní frekvenci v MHz (na kitu OMDAZZ to je 50 MHz), pokračujte dál. V kroku 4 vypněte všechny funkce (reset, locked output apod.), aby bylo PLL co nejjednodušší.




Klikejte klidně na next, next, next, až se dostanete do oddílu 3 – Output Clocks (stránka 8). Zde máte dvě možnosti: Buď 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@85 Hz, budu potřebovat frekvenci 36 MHz, kterou získám ze vstupních 50 prostým vynásobením zlomkem 18/25.




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).

## 12.5 Kalkulačka!

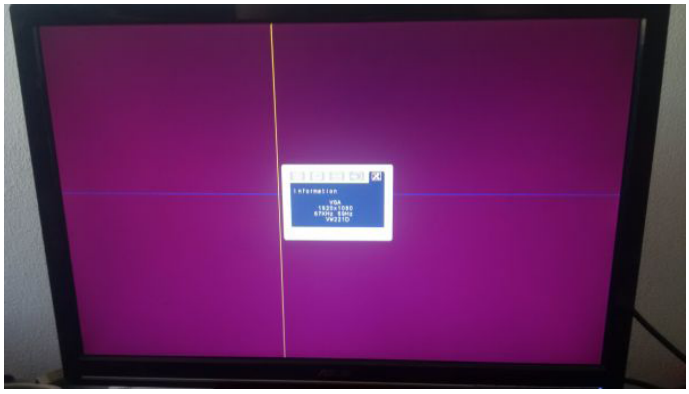
Jestli se vám motá hlava z nadmíry konstant a frekvencí, tak nezoufejte! Připravil jsem pro vás kalkulátor frekvencí pro VGA, PLL a vůbec všechno to, co jsme si teď říkali. Stačí zadat frekvenci krystalu, kterou máte k dispozici, a kliknout na Go!

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 vygeneruje vylepšenou komponentu sync.vhd (viz výše) a ukáže aktuální hodnoty průběhu signálů.

https://datacipy.cz/vga

Při svých testech jsem se odvážně pouštěl dál a dál, a nakonec jsem použil rozlišení 1920×1080@60 Hz, tedy Full HD. Pixelová frekvence je 148,5 MHz, PLL koeficient 99/32 (a opravdu, 48 * 99 / 32 = 148,5). S trochou rozechvění jsem nahrával kód do FPGA, a po několika sekundách se na monitoru objevilo:

Jak se říká: *Bylo to tam!*

## 12.6 Jednoduchý obrazec

Synchronizace je nezbytný základ, je to kostra, na který se navěsí 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 (
 clock50: in std_logic;
 vga_hs,vga_vs: out std_logic;
 vga_r: out std_logic;
 vga_g : out std_logic;
 vga_b: out std_logic
);

END VGA;

architecture main of vga is

signal vgaclk: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 (clock50, vgaclk);

process (vgaclk) begin
    if rising_edge(vgaclk) then

        -- obrazec
      if (x > 950 and x<970) then
         -- svisle barevne prouzky uprostred
         vga_r<='1';
         vga_g<='1';
         vga_b<='0';
      elsif (y > 530 and y<550) then
         -- horizontalni cara
         vga_r<='0';
         vga_g<='0';
         vga_b<='1';
      elsif (blank='0') then
         -- kde neni nic, tam je pozadi
         vga_r<='1';
         vga_g<='0';
         vga_b<='1';
      elsif (blank='1') then
         -- zatmeni
     vga_r<='0';
     vga_g<='0';
     vga_b<='0';
      end if;

    end if; --clk
end process;

end main;
```

Komponentu PLL1 mi vygeneroval výše zmíněný MegaWizard. Komponentu sync jsem si nechal vygenerovat svojí 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". Ale problematiku HDMI ještě probereme samostatně.*

# 13 Užitečné obvody

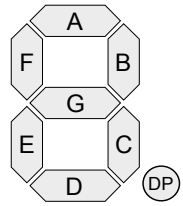
# 13 Užitečné obvody

Berte tuto kapitolu jako seznam užitečných komponent, které se vám budou hodit do vlastních konstrukcí, a zároveň si na nich procvičíte, jak psát ve VHDL. Představíme si několik elementárních obvodů pro různá rozhraní. Ponechme stranou fakt, že některé obvody FPGA mají dedikované moduly pro sériovou komunikaci, které většinou funkce I2C nebo SPI nabízejí. Důležitější je, že se právě na takových komponentách střední složitosti dají pěkně ilustrovat a procvičovat principy návrhu funkčních celků ve FPGA.

A opět platí: zdrojové kódy naleznete na doprovodném webu https://datacipy.cz/

## 13.1 Dekodér pro sedmisegmentovky

Ano, už jsem to zadával jako domácí úkol, ale opakování nikdy nezaškodí. Sedmisegmentové displeje mají osm segmentů (však co, tři mušketýři byli taky čtyři) – osmý je desetinná tečka, znaky jsou tvořeny sedmi segmenty, tradičně značenými A až G.




Dekodér navrhneme tak, aby sedmisegmentovka mohla zobrazovat hodnoty 0 až 15 (tedy znaky 0-9 a písmena A-F). Použijeme jednoduchou paměť hodnot. A uděláme si dekodér tak, aby byl kompatibilní s displejem, použitým v kitu OMDAZZ, tedy „0 = svítí, 1 = nesvítí“.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY seg7 IS
       PORT (
                 D : IN unsigned(3 DOWNTO 0);
                 --displej LED
                 --seg = segment, 0=aktivni
```

```
                segA : OUT std_logic;
                segB : OUT std_logic;
                segC : OUT std_logic;
                segD : OUT std_logic;
                segE : OUT std_logic;
                segF : OUT std_logic;
                segG : OUT std_logic
          );
END ENTITY;

ARCHITECTURE combi OF seg7 IS

TYPE decoderT IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(6 DOWNTO 0);
CONSTANT decoder : decoderT := (
    "1000000",
    "1111001",
    "0100100",
    "0110000",
    "0011001",
    "0010010",
    "0000010",
    "1111000",
    "0000000",
    "0010000",
    "0001000",
    "0000011",
    "1000110",
    "0100001",
    "0000110",
    "0001110"
            );

SIGNAL segments : std_logic_vector(6 DOWNTO 0);

BEGIN

segments <= decoder(to_integer(D));

segA <= segments(0);
segB <= segments(1);
segC <= segments(2);
```

```
segD <= segments(3);
segE <= segments(4);
segF <= segments(5);
segG <= segments(6);

END ARCHITECTURE;
```

Vstup tvoří čtyřbitové číslo D, výstup sedm segmentů.

## 13.2 Multiplexní buzení sedmisegmentového displeje

Většina sedmisegmentových displejů s více pozicemi používá takzvané multiplexní buzení. To znamená, že mají vyvedených osm vývodů pro samotné segmenty, a pak jeden vývod pro každou pozici. Postupně se aktivují pozice a k nim odpovídající segmenty, a pokud to děláte dostatečně rychle, tak to vypadá, že svítí všechny pozice najednou.

Postavme si komponentu, která obslouží čtyřmístný LED v kitu OMDAZZ. Na vstupu jí poskytneme šestnáctibitové číslo a hodinový kmitočet pro multiplexing. Uvnitř použijeme dekodér z předchozího příkladu a jednoduchý čítač 0 – 3. Výstupem bude osm segmentů a čtyři pozice (obojí aktivní v nule, protože tak je kit OMDAZZ zapojen).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY segmuxnum IS
       PORT (
                clk : IN std_logic;
                data : IN unsigned(15 DOWNTO 0);

                --fyzicky displej LED
                --seg = segment, 0=aktivni
                segA : OUT std_logic;
                segB : OUT std_logic;
                segC : OUT std_logic;
                segD : OUT std_logic;
                segE : OUT std_logic;
                segF : OUT std_logic;
                segG : OUT std_logic;
                segH : OUT std_logic;
```

```
                dig1 : OUT std_logic;
                dig2 : OUT std_logic;
                dig3 : OUT std_logic;
                dig4 : OUT std_logic

          );
END ENTITY;

ARCHITECTURE main OF segmuxnum IS

SIGNAL D : unsigned(3 DOWNTO 0) := "0000";
SIGNAL counter : INTEGER RANGE 0 TO 3 := 0;

BEGIN

PROCESS (clk) IS
        BEGIN
                IF (rising_edge(clk)) THEN
                        counter <= counter + 1;
                END IF;
 END PROCESS;

D <= data(3 DOWNTO 0) WHEN counter = 0
                ELSE
                data(7 DOWNTO 4) WHEN counter = 1
                ELSE
                data(11 DOWNTO 8) WHEN counter = 2
                ELSE
                data(15 DOWNTO 12);

dig1 <= '0' WHEN counter = 0 ELSE
                '1';
dig2 <= '0' WHEN counter = 1 ELSE
                '1';
dig3 <= '0' WHEN counter = 2 ELSE
                '1';
dig4 <= '0' WHEN counter = 3 ELSE
                '1';

decoder : ENTITY work.seg7 PORT MAP (D, segA, segB, segC, segD, segE, segF, segG);
  segH <= '1';

END ARCHITECTURE;
```

Máme dvě možnosti – buď použít čtyři dekodéry a přepínat mezi jejich výstupy, nebo naopak přepínat jednotlivé bitové pozice vstupu (dělat z něj tedy čtyři čtyřbitové *nibbles*) a přivádět až tyto části do jednoho dekodéru. Jak jsem už doporučoval v předchozím textu: *pokud to jde, používejte komponenty účelně*. Tedy jeden dekodér a přepínání vstupu!

*Pokud zvolíte příliš vysokou multiplexovací frekvenci, tak se vám může stát, že místo požadovaného uvidíte jen slabě zářící změť segmentů. Při pokusech si můžete zkusit budit multiplexor třeba základními hodinami 50 MHz, uvidíte sami! Bezpečné frekvence pro multiplexované displeje jsou v řádech jednotek kilohertzů. Bude se vám hodit generická dělička frekvencí, kterou si ukážeme vzápětí.*

## 13.3 Generická dělička kmitočtu

Často potřebujete rychle získat určitou hodinovou frekvenci ze základních systémových hodin. Je dobré připravit si generickou děličku, které zadáte dva parametry, totiž vstupní kmitočet a výstupní kmitočet, na vstup přivedete hodiny a z výstupu odebíráte požadovaný signál. Třeba ten jeden kilohertz pro buzení multiplexoru displeje.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY freqdiv IS
    GENERIC (
        freqin : NATURAL := 50_000_000;
        freqout : NATURAL := 1000
    );
    PORT (
        clk : IN std_logic;
        clkout : OUT std_logic);
END ENTITY;

ARCHITECTURE main OF freqdiv IS

signal clki:std_logic:='0'; --interni

    CONSTANT divider : NATURAL := freqin / freqout / 2;
BEGIN

    PROCESS (clk) IS
        VARIABLE counter : INTEGER := 0;
    BEGIN
```

```
        IF (rising_edge(clk)) THEN
            counter := counter + 1;
            IF (counter = divider) THEN
                counter := 0;
                clki <= NOT clki;
            END IF;
        END IF;
    END PROCESS;

 clkout <= clki;

END ARCHITECTURE;
```

## 13.4 Generátor úvodního signálu RESET

Může se hodit pro řádnou úvodní inicializaci, popřípadě i pro ošetření tlačítka RESET. Technicky nejjednodušší bude použít čítač, inicializovaný na nulovou hodnotu, který bude počítat hodinové cykly. Jakmile dopočítá do určité hodnoty, např. 50000, ukončí signál RESET a přestane počítat. Signál od vnějšího tlačítka aktivuje RESET a opět vynuluje čítač. Hodnota 50000 je zvolena tak, aby signál RESET byl dlouhý 1 milisekundu.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY reset IS
  PORT (
    clk : IN std_logic;
    resb : IN std_logic;
    reset : OUT std_logic
  );
END ENTITY;
ARCHITECTURE main OF reset IS

  SIGNAL doReset : std_logic := '1';
  CONSTANT divider : NATURAL := 50000;
BEGIN

  PROCESS (clk) IS
    VARIABLE counter : INTEGER := 0;
```

```
  BEGIN
    IF (rising_edge(clk)) THEN
      IF (doReset = '1') THEN
        counter := counter + 1;
        IF (counter = divider) THEN
          doReset <= '0';
        END IF;
      ELSE
        IF (resb = '0') THEN
          counter := 0;
          doReset <= '1';
        END IF;
      END IF;
    END IF;
  END PROCESS;

  reset <= doReset;

END ARCHITECTURE;
```

Vstup resb (Reset Button) předpokládá, že se tlačítko spíná k nule a v klidu je na něm úroveň 1. Můžete mu předřadit debouncer, který ošetří zákmity (viz níž), ale vzhledem k délce generovaného pulsu to není nutné.

Výstup reset je aktivní v log. 1.

## 13.5 Debouncer

Debouncer se hezky česky jmenuje „odstraňovač zákmitů“. Každý, kdo si zkusil třeba ovládání Arduina pomocí mechanických spínačů, zjistil, že se občas jedno stisknutí projeví jako dvě nebo tři. Důvod je prostý a spočívá v mechanickém řešení tlačítka: uvnitř je pružná kovová membrána, která se při stisknutí prohne a spojí vodivé kontakty. Jenže při prohnutí zároveň zapruží, takže ve skutečnosti třeba třikrát po sobě rychle přiskočí a odskočí, a to stačí pro vytvoření série zákmitů, které jsou sice okem a jinými smysly nedetekovatelné, ale rychlá elektronika je zaznamená.

U Arduina se tento problém nejčastěji řeší softwarově, je to totiž nejjednodušší: jakmile přijde impuls, počká se dostatečně dlouho na to, aby se tlačítko ustálilo, a pak se kontroluje znovu. Druhá možnost je použít bezzákmitová tlačítka, která jsou ale výrazně dražší.

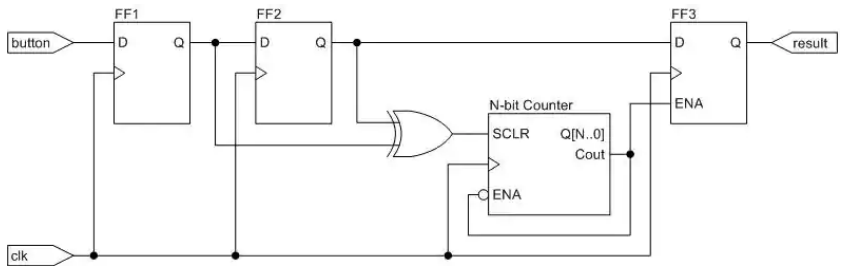
U FPGA použijeme obvodové řešení. Navrhneme si „detektor hran“, a za něj čítač, který bude počítat hodinové pulsy. Každá hrana vstupního signálu vynuluje čítač. Jakmile se vstupní signál ustálí, čítač začne počítat, a když dosáhne předem dané hodnoty, prohlásíme signál za stabilní a propíšeme jeho hodnotu na výstup. Pokud během doby čítání přijde změna vstupního signálu, čítač se vynuluje a jede se od začátku.

Čítač není problém, s těmi jsme se už setkali, otázka je, jak detekovat hranu.

## Detektor hran

Nejjednodušší způsob detekce hran je použít dva klopné obvody D kaskádovitě za sebou, čímž se vytvoří krátká zpožďovací linka, a porovnávat jejich výstupy. Pokud jsou oba signály stejné (00 nebo 11), znamená to, že je signál stabilní. Pokud se budou lišit (01 nebo 10), znamená to, že se signál změnil, změna se propsala do prvního klopného obvodu, ale do druhého ještě ne. Pro porovnávání je ideální hradlo XOR, které má na výstupu 0, pokud jsou vstupy stejné, nebo 1, pokud jsou rozdílné.

Výstup detektoru zapojíme na nulovací vstup čítače, jak jsme si popsali výše. Výstup přenosu z čítače, tedy informaci o tom, že čítač dospěl k předem zadané hodnotě, přivedeme jednak na povolovací vstup čítače (takže jakmile dosáhne maxima, už dál čítat nebude), jednak na hodinový vstup třetího klopného obvodu D, tentokrát výstupního.




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY debounce IS
  GENERIC(
    stableTime : INTEGER := 10);  -- pocet cyklu
  PORT(
    clk : IN  STD_LOGIC;
    reset : IN  STD_LOGIC; --aktivni = 0
```

```
    d : IN  STD_LOGIC;
    q : OUT STD_LOGIC);
END debounce;

ARCHITECTURE main OF debounce IS
  SIGNAL delays : STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL edge : STD_LOGIC;
BEGIN

  edge <= delays(0) xor delays(1);  --hrana

  PROCESS(clk, reset)
    VARIABLE count :  INTEGER RANGE 0 TO stableTime;
  BEGIN
    IF(reset = '1') THEN
      delays(1 DOWNTO 0) <= "00";
      q <= '0';
    ELSIF(rising_edge(clk)) THEN
      delays(0) <= d;
      delays(1) <= delays(0);
      IF(edge = '1') THEN
        count := 0;
      ELSIF(count < stableTime) THEN
        count := count + 1;
      ELSE --signal je stabilni
        q <= delays(1);
      END IF;
    END IF;
  END PROCESS;

END ARCHITECTURE;
```

V kódu je jeden generický parametr stableTime, který udává počet cyklů hodin, po který se nesmí vstupní signál změnit, aby byl prohlášen za stabilní. Dva vstupní klopné obvody jsou řešeny jako dvoubitový vektor delays(). Vstup reset je asynchronní (proto je uveden v sensitivity listu u procesu).

## 13.6 Sériové rozhraní SPI

SPI je jednoduché a široce používané sériové rozhraní. Existuje mnoho obvodů, od pamětí po

čidla, která právě toto rozhraní používají. Jde o synchronní plně duplexní sériové rozhraní typu Master-Slave, které používá dva datové signály MOSI a MISO (Master Out Slave In, resp. Master In Slave Out), signál SCK (hodinový signál, který vždy vysílá „master“) a signál SSELn (někdy též CSn – invertovaný, tj. aktivní v log. 0), který určuje, jaké zařízení má být aktivní.

Na signály MOSI, MISO a SCK může být u jednoho „masteru“ připojeno více obvodů „slave“, ale každý z nich musí mít vlastní povolovací signál SSELn.

SPI může pracovat ve čtyřech různých hodinových režimech, které se označují čísly 0 – 3, případně pomocí konstant CPOL a CPHA, neboli polarita a fáze hodin. Polarita udává klidovou hodnotu hodinového signálu, fáze říká, jestli se data čtou při přechodu z klidové do aktivní úrovně, nebo při opačném přechodu. V praxi se používají nejčastěji konfigurace 00 (klidová hodnota 0, čte se při přechodu z nuly do jedničky, tj. při vzestupné hraně) nebo 11 (klidová hodnota 1 a čte se při přechodu z aktivní do klidové úrovně, tedy opět vzestupnou hranou).

Komunikace v režimu 00 probíhá tak, že master ponechá hodiny v klidovém stavu a stáhne výstup SSELn pro slave obvod do stavu logické 0. Tím dá najevo, že hodlá s daným obvodem komunikovat.

Master začne vysílat hodinové pulsy a posílat data po výstupu MOSI tak, že při vzestupné hraně je na tomto výstupu platný bit. Posílá se od nejvyššího bitu k nejnižšímu, vždy osm bitů.

Protože je SPI plně duplexní, může ve stejný čas zároveň posílat obvod slave svá data do masteru. V praxi se spíš setkáte s tím, že nejprve master oznámí periferii, jaká data a kolik jich potřebuje, a pak slave posílá data, zatímco master řídí hodiny.

Jakmile je přenos u konce, vrátí master SSELn zpět do úrovně 1. Slave je povinen uvolnit výstup MISO, aby mohly s masterem komunikovat jiné obvody na stejné sběrnici.

SPI může přenášet data velmi rychle, až v řádech megabitů za sekundu.

### Implementace SPI MASTER

SPI MASTER bude entita s rozhraním, které musí obsahovat kromě signálů SCK, MOSI a MISO (SSELn není třeba uvažovat, výběr periferie může jít mimo a do samotné komunikace nijak nezasahuje) i rozhraní pro komunikaci se systémem uvnitř FPGA. Toto rozhraní bude zahrnovat osmibitovou datovou sběrnici, respektive rovnou dvě, jednu pro směr dovnitř, druhou pro směr ven. Kromě ní hodinový vstup, resetovací vstup, a bude se hodit i vstup CS (povolovací – enable). Pokud se smíříte s tím, že master bude používat nejběžnější hodinový režim 00, můžete oželet i vstupy CPOL a CPHA a nechat je jako generické parametry, popřípadě zadrátovat napevno. Nezbytný je i nějaký výstup „busy“, který dá vědět, že obvod stále přenáší data, a vstup,

který oznámí, že se má v přenosu dat pokračovat. Volitelný parametr určuje i dělicí poměr hodinové frekvence pro signál SCK – jednak nebudeme chtít vysílat plnou systémovou rychlostí, ale také potřebujeme nějaký čas na vnitřní operace.

Jádrem celé komponenty musí být stavový automat – v tomto případě docela jednoduchý, který používá jen dva stavy: „připraven“ a „vysílá“.

Tělo pak tvoří jeden velký proces, který sleduje změny signálů clock a reset (implementujeme asynchronní reset).

Pokud přijde signál reset, nastavíme automat do stavu ready, výstup do stavu vysoké impedance, vyčistíme přijímací buffer a nastavíme výstup „busy“ – o jeho uvolnění se postará stavový automat.

Samotný stavový automat ve stavu „ready“ nastavuje výstup „busy“ na nulu (zařízení je připraveno komunikovat), vynuluje příznak pokračujícího přenosu i vnitřní čítač děliče hodinového taktu, a sleduje vstup CS (Chip Select). Když je CS aktivní, přebere si data ze vstupní sběrnice do vysílacího bufferu, nastaví výstup „busy“, nastaví SCK na klidovou hodnotu (hodnota CPOL), nastaví si příznak čtení / zápisu podle příznaku CPHA, vynuluje počet přepnutí hodinového signálu (8 bitů, tedy 16 či 17 přepnutí, podle fáze hodin), nastaví děličku hodinového kmitočtu a přejde do stavu „komunikuje se“.

Ve stavu komunikace je signál „busy“ stále aktivní a jako hlavní úkol se generuje hodinový výstup SCK tak, že se hodinový vstup dělí zadanou konstantou. Jakmile čítač dopočítá k zadané hodnotě, pokračuje znovu od nuly, a provádí se série operací:

- mění se stav výstupu SCK
- mění se příznak vysílání/příjmu
- počítá se 16 změn
- přijímá se nebo vysílá bit do / z bufferů
- pokud je nastaven vstup „cont“ (pokračuje se ve vysílání), tak se během posledního cyklu čte další hodnota ze vstupu, jinak se ukončuje přenos, ruší se signál „busy“ a automat přechází do stavu „ready“.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
```

```
ENTITY spiMaster IS
    GENERIC (
        CLKDIV : INTEGER := 4; -- delitel hodin
        CPOL : STD_LOGIC := '0'; -- polarita
        CPHA : STD_LOGIC := '0' -- faze
    );
    PORT (
        clock : IN STD_LOGIC; --hlavni hodiny
        reset : IN STD_LOGIC; --asynchronni reset
        cs : IN STD_LOGIC; --zacatek prenosu
        cont : IN STD_LOGIC; --pokracovani prenosu
        TxD : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
              --vstup dat
        miso : IN STD_LOGIC; --master in, slave out
        sck : BUFFER STD_LOGIC; --spi clock
        mosi : OUT STD_LOGIC; --master out, slave in
        busy : OUT STD_LOGIC := '1';
               --vysilac pracuje /
               --data nejsou pripravena
        RxD : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
              --vystup dat
END ENTITY;

ARCHITECTURE main OF spiMaster IS
    TYPE FSM IS(ready, execute); --state machine
    SIGNAL state : FSM; --aktivni stav
    SIGNAL count : INTEGER;
    SIGNAL clkToggle : INTEGER RANGE 0 TO 17;
           --pocita prepnuti hodin
    SIGNAL txrx : STD_LOGIC;
           --'1' vysila se, '0' prijima se
    SIGNAL continue : STD_LOGIC; --pokracovani
    SIGNAL rxBuffer : STD_LOGIC_VECTOR(7 DOWNTO 0);
           --buffer prijimace
    SIGNAL txBuffer : STD_LOGIC_VECTOR(7 DOWNTO 0);
           --buffer vysilace
    SIGNAL lastBitRx : INTEGER RANGE 0 TO 16;
           --ktery bit je posledni
    SIGNAL cpolVectorize : std_logic_vector(0 TO 0);
           --hack kvuli aritmetice
```

```
BEGIN

    cpolVectorize(0) <= cpol; --hack kvuli aritmetice

    PROCESS (clock, reset)
    BEGIN

        IF (reset = '1') THEN --reset
            busy <= '1';
            mosi <= 'Z';
            RxD <= (OTHERS => '0');
            state <= ready;

        ELSIF (rising_edge(clock)) THEN
            CASE state IS --state FSM

                WHEN ready =>
                    busy <= '0';
                    mosi <= 'Z';
                    continue <= '0';
                    count <= 1;

                    --zacina transakce
                    IF (cs = '1') THEN
                        busy <= '1';
                        sck <= cpol;
                        txrx <= NOT cpha;
                        txBuffer <= TxD;
                        clkToggle <= 0;
-- posledni zmena hodin, ve ktere se prijimaji data
                        lastBitRx <= 15 +
                  to_integer(unsigned(cpolVectorize));
                        state <= execute;
                        count <= CLKDIV;
         -- aby zmena probehla hned
                    ELSE
                        state <= ready;
                    END IF;

                WHEN execute =>
                    busy <= '1';
```

```
 --pokud je delicka kmitoctu na nejvyssi hodnote
               IF (count = CLKDIV) THEN
                   count <= 1; --nulujeme delicku
                   txrx <= NOT txrx;
 --pocet prepnuti hodinoveho pulsu
                   IF (clkToggle = 17) THEN
                       clkToggle <= 0;
 --uz jsme skoncili
                   ELSE
                       clkToggle <= clkToggle + 1;
                   END IF;

 --je potreba zmenit SCK?
                   IF (clkToggle < 16) THEN
                       sck <= NOT sck;
                   END IF;

--prijem
                   IF (txrx = '0' AND clkToggle < lastBitRx + 1) THEN
                       rxBuffer <= rxBuffer(6 DOWNTO 0) & miso;
                   END IF;

                   --vysilani
                   IF (txrx = '1' AND clkToggle < lastBitRx) THEN
                       mosi <= txBuffer(7);
                       txBuffer <= txBuffer(6 DOWNTO 0) & '0';
                   END IF;

 --posledni bit, ale pokracuje se
                   IF (clkToggle = lastBitRx AND cont = '1') THEN
                       txBuffer <= TxD; --nacist novou hodnotu
                       clkToggle <= lastBitRx - 17;
                       continue <= '1';
                   END IF;

 --prenos byte skoncil, ale pokracuje se
                   IF (continue = '1') THEN
                       continue <= '0';
                       busy <= '0';
 --puls na vystupu busy dava signal
```

```
      --pro dalsi prenos
                           RxD <= rxBuffer;
                       END IF;

      --konec transakce?
                       IF ((clkToggle = 17) AND cont = '0') THEN
                           busy <= '0'; --busy zrusit
                           mosi <= 'Z';
                           RxD <= rxBuffer;
      --precteny byte
                           state <= ready;
      --zpet do stavu ready
                       ELSE
      --jeste neni konec?
                           state <= execute;
      --zustavame ve stavu execute
                       END IF;

                   ELSE
      --delicka kmitoctu jeste nedopocitala k maximu?
                       count <= count + 1;
      --zvysime jeji hodnotu
                       state <= execute;
      --a zustavame ve stavu execute
                   END IF;

              END CASE;
          END IF;
      END PROCESS;
END ARCHITECTURE;
```

## Implementace SPI Slave

Zatímco master můžeme navrhnout poměrně univerzálně, u slave zařízení tomu tak není. Hodně záleží na typu konkrétního zařízení, a protože není pevně daný protokol, záleží jen na návrháři, jak jej pojme. Často se SPI slave zařízení navrhují jako „paměti“, kdy master posílá adresu, ze které chce číst nebo do které chce zapisovat, a následuje přenos dat. Popřípadě se nejprve posílá „příkaz“, například „čtení konfigurace“ nebo „reset zařízení“, a pak následují potřebná data (třeba u sériových pamětí).

Případů, kdy budete ve VHDL implementovat slave SPI, bude pravděpodobně výrazně méně než těch opačných. Proto nebudeme řešit konkrétní implementaci rozhraní SPI Slave, ta se bude lišit podle typu zařízení, ale jen si řekneme hrubé rysy:

- Zařízení se synchronizuje příchozími hodinovými pulsy (SCK). Případné čtení či zápis dat bude tedy vyžadovat synchronizaci s vnitřními hodinami, a tedy řešení problému hodinových domén.

- „Problém hodinových domén" lze vyřešit u jednoduchých zařízení, které vyžadují jen několik „vnitřních registrů". V takovém případě je možné implementovat tyto registry v rámci SPI Slave rozhraní, a pro zbytek zařízení vystavit hodnotu těchto registrů.

- Nemusíte pravděpodobně řešit konfiguraci CPOL a CPHA – můžete ji zadat napevno, a nic asi nepokazíte, když zvolíte „00".

## 13.7 Rozhraní I²C

Druhé často používané rozhraní pro komunikaci s periferiemi nese označení I²C. Je lehce podobné rozhraní SPI s několika rozdíly:

- Používá pouze dva signály, jeden pro hodiny (SCL), druhý pro data (SDA).

- K jejich buzení se používají budiče s otevřeným kolektorem, tj. je nutné připojit pull-up rezistory.

- Každé zařízení má svou sedmibitovou adresu (v pozdějších revizích desetibitovou). Na sběrnici tak můžete připojit teoreticky přes 100 zařízení, ale každé z nich musí mít jinou adresu.

- Díky tomu se ušetří vodiče pro výběr zařízení.

- Sběrnice je multi-master, to znamená, že je možné, aby ji řídilo více zařízení (nikoli najednou).

Na druhou stranu je sběrnice I²C výrazně pomalejší (často se komunikuje s hodinovou frekvencí 400 kHz, ale existují periferie, které tak rychle komunikovat nedokáží, proto se používají i rychlosti 10 kHz nebo 100 kHz. Novější protokol nabízí naopak i rychlosti vyšší).

Komunikace je standardizovaná, dobře dokumentovaná a známá. Signály SCL a SDA jsou v klidu v 1. Zařízení master oznámí začátek komunikace tím, že na sběrnici vytvoří takzvanou „startovní podmínku" (start condition), totiž stáhne signál SDA k nule a poté stáhne i signál SCK k nule.

Po startovní sekvenci vyšle master sedmibitovou adresu (od nejvyššího bitu), vždy tak, že při náběžné hraně hodin je hodnota platná a zařízení ji vzorkují. Osmým bitem úvodní komunikace je bit, který udává, zda se čte (1), nebo zapisuje (0). Pak master uvolní datový signál a vyšle ještě jeden hodinový puls, během něhož naslouchá.

Všechna slave zařízení na sběrnici mezitím čtou sedmibitovou adresu. Pokud se neshoduje s jejich adresou, nadále už do komunikace nezasahují, čekají na konec aktuální komunikace, a pak se proces opakuje.

Pokud zařízení rozpozná svou adresu, odpoví tím, že v devátém hodinovém pulsu, kdy master neřídí datový signál, samo pošle bit 0 – potvrzovací bit ACK. Master, který v tu chvíli naslouchá, podle toho zjistí, že adresované zařízení je připojené a připravené.

V tu chvíli je navázáno spojení. Master oznámil zařízení s danou adresou, že s ním komunikuje, zařízení ohlásilo připravenost, a následuje další komunikace.

Ta záleží na typu zařízení. Opět platí, že nejčastější model je takový, že se zařízení tváří jako „paměť“, takže master pošle adresu „registru“, a pak buď pošle data, která chce zapsat, nebo naopak čeká na data ze zařízení. Po každém tomto kroku potvrzuje zařízení, které data přijímalo, že data přijalo v pořádku vysláním bitu ACK.

*Když vysílá master a slave přijímá, posílá ACK slave. Pokud naopak master čte a slave vysílá požadovaná data, potvrzuje příjem master.*

Po ukončení přenosu pošle master opačnou sekvenci než na začátku, tzv. „Stop condition“. Nejprve stáhne SDA k nule (SCL v nule je po konci hodinového pulsu). Poté uvolní SCL (zařízení detekují náběžnou hranu hodin, po níž by se SDA už neměl měnit), a až poté uvolní SDA. Tímto postupem zařízení poznají, že šlo o konec komunikace.

Zápis (tedy proces, kdy master posílá data zařízení slave), je jednoduchý a probíhá tak, jak bylo popsáno. Čtení dat, tedy přenos v opačném směru, je o něco komplikovanější. Většinou se do zařízení pošle nejprve adresa, ze které se má číst, pak následuje opět start, pošle se opět sedmibitová adresa a nejnižší bit = 1 (tedy příznak pro čtení), a od té chvíle posílá data slave. Master může po každém přečteném byte poslat signál ACK, tím se hodnota interního čítače adres ve slave zařízení zvýší o 1 a slave pošle obsah paměti na další adrese. Tak lze číst více bytů naráz. Po posledním bitu master pošle NACK (not ACK, nepotvrzeno, tj. bit s hodnotou 1) a nastaví stop condition.

*Píšu o „adresách paměti" a podobně proto, že to je nejčastější model přístupu k těmto zařízením. I pokud se nejedná opravdu o paměti. Různá čidla či hodiny reálného času používají stejný model a programátor k nim přistupuje jako k paměti či k adresované sadě registrů, kde každý z registrů má nějakou funkci.*

### I²C slave

Můžeme zvolit dva různé způsoby implementace. Jeden je ten, že přenos časuje vnější signál SCL, druhý ten, že přenos řídí vnitřní hodiny (třeba těch 50 MHz) a oba signály jsou filtrovány a s touto frekvencí vzorkovány. U druhého způsobu je o něco složitější detekce start a stop stavu – z popisu vyplývá, že tyto stavy nastávají, pokud se stane něco, *a potom* něco jiného. Pokud se to *potom* stane příliš rychle, může se stát, že to vlivem samplování splyne v jednu současnou změnu, což je špatně.

Pro I²C Slave platí totéž, co jsme si říkali u SPI Slave: zařízení jsou různorodá a univerzální rozhraní nemůže implementovat vše. Jako rozumný kompromis se jeví generické rozhraní se zadanou sedmibitovou adresou, které reaguje na start condition, rozpozná správnou adresu a pak přijímá nebo vysílá data. Uvnitř zařízení musí být ale mechanismus, který ví, kolik byte se má načíst a jak na ně reagovat – buď procesor, nebo alespoň stavový automat.

Rozhraní může ještě implementovat automatické posílání ACK bitů, případně jejich kontrolu po vysílání dat.

### I²C master

Master zařízení může být snáze zkonstruováno jako univerzální – většinou se už počítá s tím, že jej bude ovládat procesor. Můžeme jej navrhnout buď jako naprosto transparentní rozhraní, kterému pouze zadáváte, jestli má poslat start sekvenci, stop sekvenci, popřípadě data, nebo jestli má data přijmout. Můžeme jej ale navrhnout i o něco sofistikovaněji, s vlastní adresní sběrnicí.

Při návrhu je potřeba přivést do rozhraní systémový hodinový kmitočet, ze kterého pak master vnitřně vygeneruje standardní přenosové kmitočty pro I²C sběrnici, tj. 10, 100 nebo 400 kHz, případně víc u nových zařízení.

Dále budete potřebovat vybavovací vstup, CS (chip select), kterým začne přenos dat (pokud se stane aktivní, zařízení vyšle start condition, jakmile přejde zpět do 0, master dokončí aktuální operaci a pošle stop condition).

Nutné jsou datové sběrnice pro vstup a výstup dat, signál „busy", který oznámí, že probíhá operace, a výstup „error", který signalizuje, že došlo k chybě a nebyl přijat správně ACK bit.

Nakonec jsou zapotřebí samotné signály SCL a SDA. Oba jako obousměrné, s otevřeným kolektorem. Důvod, proč je dobré mít i hodiny s otevřeným kolektorem, je ten, že některá slave zařízení používají takzvanou techniku „clock stretching“, když potřebují, aby master počkal s další operací. Pokud je nutné „natažení času“, stáhne slave signál SCL k nule a dá tak masteru najevo, že je potřeba počkat.

*Zdrojový kód je velmi rozsáhlý, proto jej nechávám zájemcům k dispozici online v příkladech ke knize.*

## 13.8 Připojení SD karty

Karty formátu SD (popřípadě novější SDHC, SDXC) jsou, a ještě asi několik let budou, nejvýhodnějším řešením pro ukládání velkých objemů dat. Mají malé rozměry, snadno se připojují a jejich poměr cena / kapacita z nich dělá velmi výhodné médium, na druhou stranu mají nevýhodu v tom, že použitá technologie časem degraduje a mají jen omezený počet zápisů.

Karty jsou dostupné ve dvou rozměrech, totiž „velké“ SD a pak MicroSD. Vývodově jsou totožné a z „malé“ karty uděláte pomocí redukce snadno „velkou“. Záleží tedy na vás, jaký rozměr vám bude vyhovovat a jaký konektor pro vás bude dostupný.

SD karta má 9 pinů, MicroSD osm pinů (signály jsou stejné, i když jinak rozvedené; u SD karty jsou vyhrazené dva piny pro zem, u MicroSD jen jeden).

SD karta je oblíbená i proto, že ji lze používat v módu SPI. V režimu SPI využijete čtyři signálové piny, jak je u této sběrnice zvykem: DI, DO, SCK a CS. Všechny jsou jednosměrné, nepotřebujete tedy pull-up rezistory, a pokud máte v návrhu jiné SPI zařízení, použijete stejný interface a stejný postup.

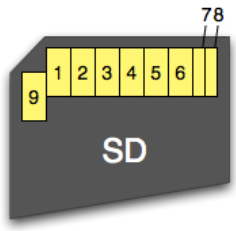
Nativní režimy komunikace jsou dva, jednobitový a čtyřbitový. U této komunikace jsou všechny signály obousměrné (kromě hodin), proto potřebujete pull-up rezistory, které umožňují připojení budičů s otevřenými kolektory (naštěstí FPGA mívají možnost zapnout interní). Používá se jeden hodinový signál (CLK), jeden signál pro přenos příkazů (CMD) a jeden nebo čtyři datové signály DAT0-DAT3. Tento režim komunikace si popíšeme.

Komunikace probíhá tak, že hostitel pošle příkaz, který má karta vykonat, a poté čte nebo zapisuje data, podle konkrétního příkazu. Příkazy mají 48 bitů: začínají nulovým, končí jedničkovým, po start bitu následuje jedničkový bit, který indikuje příkaz, pak 6 bitů samotného příkazu, 32bitový argument (např. adresa nebo jiné parametry) a před stop bitem přijde ještě 7bitový

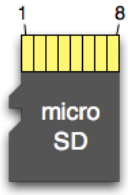
CRC kód, spočítaný ze start bitu, příznaku příkazu, ID příkazu a argumentu.

Pokud jde o příkaz, který např. zjišťuje formát nebo typ karty, má odpověď rovněž 6 byte a téměř totožný formát, s tím rozdílem, že druhý bit přenosu (po start bitu) je 0.

Při přenosu dat se typicky posílá celý blok, tj. 512 byte. To je 512x8=4096 bitů. K nim se přidává nulový start bit, jedničkový stop bit a 16bitový CRC (počítaný pouze z dat, nikoli ze start bitu jako u příkazu), takže celková délka přenášených dat je 4114 bitů.




| Pin | SD | SPI |
|---|---|---|
| 1 | CD/DAT3 | CS |
| 2 | CMD | DI |
| 3 | VSS1 | VSS1 |
| 4 | VDD | VDD |
| 5 | CLK | SCLK |
| 6 | VSS2 | VSS2 |
| 7 | DAT0 | DO |
| 8 | DAT1 | X |
| 9 | DAT2 | X |




| Pin | SD | SPI |
|---|---|---|
| 1 | DAT2 | X |
| 2 | CD/DAT3 | CS |
| 3 | CMD | DI |
| 4 | VDD | VDD |
| 5 | CLK | SCLK |
| 6 | VSS | VSS |
| 7 | DAT0 | DO |
| 8 | DAT1 | X |

### Inicializace SD karty

Inicializace probíhá v několika krocích. Při ní by neměla frekvence hodinového signálu překročit 400 kHz, později ji můžete zvýšit.

1. Nechte CMD i DAT ve stavu log. 1 a pošlete 74 cyklů hodin.

2. Pošlete příkaz 0 (GO_IDLE_STATE)

3. (Pro práci se SDHC kartami) Pošlete příkaz 8 (SEND_IF_COND). Pokud je k dispozici

SDHC karta, odpoví paketem s hodnotou 3Fh a hodnotou FFh jako CRC. Tímto příkazem se nastaví i napájecí napětí na 3,3 V.

4. Pošlete příkaz 55 (CMD55, APP_CMD). Karta vrátí rovněž hodnotu 55.

5. Pošlete příkaz 41 (SD_SEND_OP_COND). Karta odpoví opět 3Fh/FFh jako v kroku 3. Pokud je ale 39. bit (první bit argumentu) nulový, znamená to, že je karta zaneprázdněna a musíte opakovat kroky 4 a 5, dokud nebude tento bit jedničkový.

6. Pošlete příkaz 2 (ALL_SEND_CID). Karta vrátí svoje parametry, jako je výrobce, typ, číslo revize apod.

7. Pošlete příkaz 3 (SEND_RELATIVE_ADDR). Karta vrátí jako odpověď svou relativní adresu (RCA), kterou použijeme v následujícím kroku.

8. Pošlete příkaz 7 (SELECT_CARD) s RCA. Tím je karta připravena ke komunikaci a přenosu dat.

Podrobnosti o protokolu SD karet jdou daleko za rámec této knihy, zájemce o detaily proto odkážu na materiály k těmto kartám.

http://wiki.seabright.co.nz/wiki/SdCardProtocol.html

https://www.sdcard.org/downloads/pls/

Navrhnout řadič, který umí implementovat celý protokol transparentně je možné, ale pravděpodobně spíš využijete jen modul rozhraní na fyzické úrovni, který bude umět serializovat a deserializovat data a příkazy, popřípadě počítat CRC.

## 13.9 Generátor parity

Parita binárního čísla vyjadřuje, jestli je počet jedniček v tomto čísle lichý, nebo sudý. Nejčastěji se používá k velmi jednoduchému ověření správnosti přenosu. Kromě (například) osmi bitů dat se přenáší i devátý bit, zvaný paritní.

Paritní bit má takovou hodnotu, aby celkový počet jedniček ve všech devíti bitech byl sudý (sudá parita) nebo lichý (lichá parita). Například binární osmibitová hodnota 00h má osm nul, tedy žádnou jedničku. Nula je v tomto případě sudé číslo, takže paritní bit pro sudou paritu bude 0, pro lichou 1 (aby počet jedniček byl lichý). Binární hodnota DAh (1101_1010) obsahuje pět jedniček, tedy lichý počet. Paritní bit pro sudou paritu bude tedy 1 (šestá jednička, tedy sudý počet), pro lichou 0.

To, jestli se kontroluje sudá nebo lichá parita, je potřeba předem určit. Pokud jedna strana přenosu bude vysílat třeba lichou a druhá strana očekávat sudou, bude každý přenesený bit hlásit chybu.

Sudou paritu nejjednodušeji spočítáme pomocí XORu všech bitů, jednoho po druhém. Pokud si bity označíme písmeny A-H, tak:

Sudá parita = A xor B xor C xor D … xor H

Postupný XOR je u osmi bitů ještě únosný, ale počítat paritu třeba 128bitových čísel už by vedlo k výraznému zpoždění. Můžeme tedy XORovat postupně:

Sudá parita = ((A xor B) xor (C xor D)) xor ((E xor F) xor (G xor H))

Počet hradel zůstal nezměněn, ale signál nejde přes sedm stupňů postupně, ale jen třemi stupni, a každý bit urazí stejnou cestu, než se projeví na výsledku.

Volbu liché / sudé parity nejjednodušeji zařídíme tak, že výsledek XORujeme s 0, pokud chceme paritu sudou, nebo s 1, pokud chceme paritu lichou. Tak můžeme přepínat sudou / lichou paritu fyzickým signálem.

## 13.10 Připojení PS/2

Klávesnice s rozhraním PS/2 jsou dodneška populární v amatérských konstrukcích. Důvodem je, že jsou jednoduše použitelné.

Rozhraní PS/2 je obyčejné sériové rozhraní s komunikací po dvou vodičích, kde jeden je vyhrazen pro data a druhý pro hodinový signál. Jediné, co člověka při implementaci tohoto rozhraní zarazí, je fakt, že hodinový signál negeneruje ta část, ke které se klávesnice připojuje, tedy např. procesor, ale sama klávesnice.

Komunikace může probíhat navíc obousměrně (tak hostitelské zařízení například přepíná stav indikačních LED), proto používají zařízení připojení s otevřeným kolektorem a pull-up rezistory (které naštěstí můžeme zapnout v konfiguraci FPGA).

Klávesnice PS/2 vyžaduje napájecí napětí 5 voltů a pracuje s pětivoltovou logikou. Musíte proto buď použít převodní členy (např. z tranzistoru MOSFET), nebo alespoň rezistorový dělič.

### Protokol

V klidu jsou obě zařízení odpojena a na datovém i hodinovém vodiči je tedy úroveň log. 1 (díky pull-up rezistorům). Komunikaci začíná klávesnice, a to tím, že na datovou sběrnici pošle start bit s hodnotou 0 a vyšle hodinový puls (0 – 1). Hostitelské zařízení by v tu chvíli mělo začít přijímat data a vzorkovat je při každé sestupné hraně signálu clock. Klávesnice po úvodním start-bitu pošle osm datových bitů (od nejnižšího), pak paritní bit (lichá parita, tedy data spolu s paritním bitem musí mít lichý počet jedniček) a jedničkový stop-bit. Jde vlastně o sériový přenos, známý z implementace UART, s paritou a externími hodinami.

Klávesnice posílá hodinové signály s frekvencí mezi 10 a 16,7 kHz, tedy s periodou 60 až 100 μs. Pokud hodinový signál zůstane přes 55 μs v neaktivním stavu (log. 1), má se zato, že přenos skončil, klávesnice je ve stavu „idle“, tj. nečinná, a veškerá dosud přijatá data je potřeba zahodit a vynulovat.

Přenášená data jsou tzv. „scan kódy“, tedy kódy, které udávají, jaká událost nastala, jestli šlo o stisknutí klávesy nebo její puštění. Například kód 1Ch oznamuje stisk klávesy A, kódy F0h a 1Ch oznamuje puštění klávesy A, kódy E0h a 70h oznamují stisk klávesy INSERT... Konkrétní přiřazení znaků jednotlivým scan kódům už je mimo rámec této kapitoly.

### Zapojení rozhraní

Dané signály jsou ps2_data a ps2_clk. K těmto signálům musíme připojit externí klávesnici. Budeme potřebovat i interní hodiny clk, které poslouží k časování všech částí rozhraní. Výstup bude tvořit osmibitová datová sběrnice a signál *ready*.

Vstupní signály nejprve synchronizujeme s hodinami clk a ošetříme debouncerem. Poté je můžeme použít k buzení jedenáctibitového posuvného registru (8 bitů data + 1 bit parity + 1 stop bit + 1 start bit = 11). V tomto registru budeme mít jednak výstupní data (bity 1 až 8), ale i nezbytná data pro kontrolu rámce: bit 0 musí být 0, bit 10 musí být 1 a bit 9 musí spolu s datovými bity dát lichou paritu.

Pokud data neobsahují chybu a na vstupu nastal klid (stav idle), nastavíme výstup ready, kterým oznámíme, že přijatá data jsou k dispozici a připravena ke čtení.

### Vysílání dat

Ukázali jsme si, jak PS/2 přijímá data, která posílá klávesnice. Podobný postup je možné použít v případě, že chce hostitelské zařízení naopak poslat data do klávesnice.

Postup je takový, že nejprve hostitel stáhne hodinový signál k nule alespoň na 100 μs. Poté hodi-

nový signál uvolní a pošle na datovou sběrnici hodnotu 0. Klávesnice začne generovat hodinové pulsy. Při každé sestupné hraně pošle hostitelské zařízení jeden bit dat (od nejnižšího), který si klávesnice při vzestupné hraně převezme. Po odeslání osmi bitů dat a paritního bitu uvolní hostitelské zařízení datovou sběrnici. Pull-up rezistor tak na datovém vodiči vytvoří log. 1, což klávesnice při dalším hodinovém pulsu přečte jako stop bit. Pokud vše proběhlo v pořádku, pošle klávesnice zpět bit s hodnotou 0 jako potvrzení (ack), že data byla přijata. Nato se opět sběrnice uvolní.

Rozšíření přijímače na transceiver (transmitter+receiver, vysílač a přijímač) znamená rozšířit posuvný registr o možnost nahrání dat vnějším signálem a zavést výstup z posuvného registru na vstup ps2_data. Zároveň je potřeba oba signály pro ps2 pojmout jako obousměrné, s možností přepínání směru.

### Připojení myši

K transceiveru PS/2 můžeme připojit nejen klávesnici, ale i myš. Myš komunikuje stejně jako klávesnice a posílá vždy trojici byte:

| | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|
| **Byte 1** | Y ov | X ov | Y sg | X sg | 1 | MB | RB | LB |
| **Byte 2** | Posun v ose X | | | | | | | |
| **Byte 3** | Posun v ose Y | | | | | | | |

Druhý a třetí bit udává posun myši od posledního vysílání, v ose X a v ose Y. V prvním byte je informace o směru posunu (X sg a Y sg), informace o stisknutí tlačítek (middle, right, left) a informace o přetečení ve směru X či Y (overflow) – to pokud se myš posunula o vzdálenost, která se nevejde do osmi bitů.

Na začátku je potřeba myš inicializovat speciální posloupností příkazů. Nejprve se posílá reset (FFh), na který myš odpoví potvrzením (FAh). Pak spustí self test a na jeho konci pošle buď informaci o tom, že vše je v pořádku (AAh), nebo že nastala chyba (FCh). Pokud bylo vše v pořádku, pošle myš své Device ID (00h). Od té chvíle začne myš sbírat informace o pohybu. Bohužel, nezačne je posílat, posílání je zakázané. Nejprve musí hostující zařízení poslat povolovací příkaz F4h, na který myš odpoví potvrzením (FAh). Pak teprve vysílá data, jak jsme si ukázali výše.

Zařízení může buď použít obecný transceiver a řídit výše popsaný přenos samo, nebo můžeme logiku inicializace zapracovat do transceiveru a vystavovat pouze načtená data.

## 13.11 SDRAM

Velmi často se u lepších a dražších kitů setkáte s dynamickou pamětí SDRAM. Je to pochopitelné – jsou levné a nabízejí velkou kapacitu, v řádech megabajtů, tak proč je tedy neintegrovat?!

Pokud se rozhodnete tuto paměť využít, záhy zjistíte, že to není tak jednoduché, jako používat třeba statické paměti 62256. Tam je zapojení jednoduché. Paměť má adresovou sběrnici a datovou sběrnici, tři řídicí signály (CS, OE a WR) – a tím to hasne. Pošlete adresu, na datové sběrnici přečtete data... Tak fungují statické paměti.

Dynamické paměti to mají trochu složitější. Ty, co se kdysi používaly, byly v docela malých pouzdrech, přestože měly kapacitu třeba 16 kilobitů (tj. 14 adresních vstupů) – typ 4116, používaný hojně v domácích počítačích 80. let. Vtip byl v tom, že měly organizaci 16384 x 1 bit, takže jste jich potřebovali do počítače osm vedle sebe. Druhý zádrhel byl v tom, že adresa (14 bitů) se neposílala naráz, ale nadvakrát, jako 2 x 7 bitů. Nejprve se posílal výběr řádku matice (7 bitů), k tomu se aktivoval signál RAS (Row Address Select), pak se posílal výběr sloupce v rámci daného řádku (opět 7 bitů) a k tomu aktivoval signál CAS (Column Address Select), a pak se četla data (popřípadě zapisovala).

Samozřejmě to vyžadovalo trochu péče, protože třeba u čipů 4116 jste museli dodržet například to, že CAS přišel po RAS nejdřív po 20 a nejpozději po 50 nanosekundách. Když jste měli systém s hodinovým kmitočtem 2 MHz, tak jeden puls hodin trval 500 ns, takže se k časování nedal použít. Řešilo se to různými monostabilními klopnými obvody či RC členy.

Data se na výstupu objevila taky až za *nějaký čas*. Technická dokumentace uvádí, že od aktivace signálu RAS to bylo maximálně 150 ns. I s tímto jste tedy museli, coby návrhář, počítat a obvod tomu přizpůsobit.

A k tomu celému ještě přistupovala nutnost provést občerstvení, neboli refresh. Museli jste se postarat o to, abyste nějak přistoupili ke všem 128 řádkům paměti nejpozději během dvou milisekund. Nemuseli jste je kompletně číst, stačilo je jen aktivovat signálem RAS.

Důvodem toho všeho byla samotná konstrukce dynamických pamětí. Na rozdíl od statických, kde jsou informace zapamatované v klopném obvodu, je u dynamických pamětí k ukládání informace použit malý kondenzátor (respektive parazitní kapacita, vybudovaná v křemíkové struktuře).

Paměť k informacím přistupuje po řádcích – vybraný řádek si načte do vnitřního 128bitového bufferu (tento krok se nazývá *aktivace*), čímž se, jen tak mimochodem, odčerpá část náboje z paměťových buněk. Po přečtení dat se opět řádek z bufferu uloží do paměťových buněk, čímž se náboj opět obnoví (tento krok se nazývá *precharge*, což je trochu matoucí, protože se odehrává na konci, nikoli *před něčím*).

Do skládačky chybí už jen jediný dílek: kondenzátory se vybíjejí samovolně, proto bylo potřeba zajistit, aby každý řádek paměťové matice prošel alespoň jednou za 2 milisekundy tímto cyklem – a k tomu právě slouží refresh.

Takže suma sumárum: pomalé, komplikované, a u jmenovaného čipu 4116 dokonce vyžadující tři napájecí napětí (+12, +5 a -5 voltů), zato s mnohem větší kapacitou, než umožňují statické RAM, a s nižší cenou. Což mimochodem platí dodneška, proto se používají jako operační paměť u většiny počítačů (a kvůli jejich výrazné pomalosti se používají většinou s nějakou vyrovnávací pamětí – cache).

Samozřejmě dnešní dynamické paměti jsou už výrazně přívětivější. Napájení vyžadují standardní, refresh si dokáží zajistit samy, jen multiplexovaná sběrnice a pomalost přetrvává.

SDRAM označují „synchronní dynamické RAM“. Většinou se v kitech setkáte se zástupci starší generace SDR SDRAM (to první SDR j Single Data Rate). Modernější DDR (Double Data Rate) už znáte ze současných PC. Paměti SDRAM se používaly ve známých „modulech DIMM“. Na některých byste určitě našli i čip Hynix HY57V641620FTP-H – a tentýž čip je součástí kitu Cyclone IV (OMDAZZ), který jsem doporučoval v úvodu pro pokročilejší pokusy.

*Obvod by měl být kompatibilní např. s obvodem W986416CH od výrobce Winbond. Podobná paměť, jen rychlejší, je NEC D4564163 – její datasheet je nejobsáhlejší a podává podrobný popis toho, jak s pamětí pracovat.*

Tento čip nabízí 4 paměťové banky, z nichž každá má kapacitu 1M x 16 bitů, tedy 4M šestnáctibitových slov (anebo 8 megabajtů či 64 megabitů, jak chcete). Suffix -H znamená, že jde o čip s hodinovou frekvencí 133 MHz.

Na čipu naleznete většinu známých a očekávatelných signálů – 12bitovou adresní sběrnici A0-A11, signály /RAS, /CAS a /WE, 16bitovou datovou sběrnici DQ0 – DQ15, dva signály pro výběr jedné ze čtyř bank BA0, BA1, signál /CS (Chip Select), hodinový vstup CLK (všechny vstupy jsou vzorkovány náběžnou hranou tohoto signálu) a signál CKE, který umožňuje paměť uvést do stavu spánku a nízké spotřeby.

*Někteří výrobci označují signál pro výběr banky jako BS (Bank Select).*

Kromě těchto signálů má paměť i dva vstupy LDQM a UDQM, které „maskují“ datovou sběrnici a umožňují přistupovat pouze k hornímu (UDQM) či dolnímu (LDQM) byte z dvoubytového slova.

Hodinový signál může být nejvýš 133 MHz, minimálně však 1 MHz.

Při připojování k FPGA se používá takzvaný „SDRAM controller“, což je obvod, který má dvě sady sběrnic. Z jedné strany jsou vyvedené signály pro fyzický obvod SDRAM, z druhé strany, ze strany vnitřního systému, se simuluje „běžná paměť“ – tedy datová a plná adresová sběrnice, signály pro čtení, zápis a pro výběr obvodu.

Uvnitř SDRAM controlleru je opět stavový automat, který řídí všechny nutné operace. Je řízen signály ze systémové sběrnice a stará se o správné řízení paměti, o správné časování (je třeba dodržovat posloupnost a časování signálů RAS a CAS) i o vyvolání občerstvovací rutiny.

Pokud nevyžadujeme žádné „speciální operace“, jako rychlé čtení velkého množství dat (burst), postačí poměrně jednoduchý řadič.

*Samozřejmě existují i vysoce univerzální a konfigurovatelné řadiče v knihovnách IP komponent...*

Pokud budete ve FPGA používat osmi- či šestnáctibitové mikroprocesorové konstrukce, vystačíte si s jednodušším řadičem. Navíc v takových konstrukcích většinou budete od paměti požadovat náhodný přístup k různým adresám, takže rychlý přenos stěží využijete natolik, aby bylo zapotřebí jej implementovat.

Z datasheetu se dozvíme, že refresh je zapotřebí vyvolat alespoň jednou za 15 mikrosekund (4096 za 64 ms). Signál pro refresh necháme na vnějším systému – může jej zavolat vždy, když nebude potřeba přístup k paměti.

### Práce se SDRAM

SDRAM ukládají data v bankách. Každá banka je uspořádána jako matice X řádků krát Y bitů (sloupců). Naše paměť má 4 banky, každá má 4096 řádků a 256 sloupců. Adresa řádku má tedy 12 bitů, adresa sloupce 8 bitů, adresní vstup A10 se u adresy sloupce používá jako tzv. „precharge flag“, tedy příznak automatického vyvolání cyklu *precharge*.

Na rozdíl od SRAM či EEPROM jsou paměti SDRAM z programátorského hlediska bližší třeba sériovým pamětem – musíte jim posílat „příkazy“, co mají dělat, tedy například „číst od zadané adresy“. Příkazy se zadávají jako kombinace hodnot na vstupech CS, RAS, CAS a WE. Například NOP (nic se neděje) má hodnotu 0111 (aktivovaný CS, zbytek neaktivní), REFRESH je 0001 (aktivní CS, RAS i CAS)

Normální operace vypadá tak, že paměť aktivujete, tj. pošlete příkaz a adresu, pak následuje

samotné čtení či zápis, a cyklus je ukončen operací „precharge“. Čas od aktivace do znovunabití (tak si dovolím překládat *precharge*) je pro každou banku omezený, a tak se používají techniky pro efektivní střídání bank, aby byl přístup co nejrychlejší. U jednoduchého řadiče to není zapotřebí.

Důležité je dbát na to, že existují rychlostní limity – v první řadě čas od aktivace do přístupu k datům ($t_{RCD}$, RAS to CAS Delay – u naší paměti 20 ns), minimální čas od znovunabití k aktivaci ($t_{RP}$, RAS Precharge – 20 ns), a v neposlední řadě čas mezi dvěma aktivacemi ($t_{RRD}$, RAS to RAS Delay – 15 ns). Samotná operace (čtení, zápis nebo refresh) trvá 63 ns.

Při občerstvování se nemusíme starat o posílání konkrétní adresy, o to se postará SDRAM sama, stačí jen posílat pravidelně příkazy REFRESH, a to alespoň jednou za 0,015 ms.

Po zapnutí napájení je zapotřebí paměť inicializovat. Různé paměti mají lehce odlišné postupy, ale v zásadě se drží tohoto scénáře:

- Nastavit xDQM a CKE signály do log. 1, poslat signál NOP a počkat 0,2 ms
- Nabít všechny banky (precharge)
- Vykonat osm cyklů refresh
- Nastavit řídicí registr (Mode Register)
- Provést opět osm občerstvovacích cyklů

Řídicí registr určuje některé parametry, jako např. množství dat, přenášené během jednoho čtecího příkazu (burst size) nebo parametr *CAS latency* (2 nebo 3).

Stavový automat (FSM) tedy potřebuje ošetřit stavy při inicializaci (INIT_WAIT, INIT_PRECHARGE, INIT_REFRESH1, INIT_MODE, INIT_REFRESH2), pak základní stav IDLE, stav REFRESH, a stavy, potřebné při přístupu k datům: ACTIVATE (nastavení RAS a příkazu čtení), RCD (čekání na výběr řádku), RW (samotné čtení či zápis), RAS1 (čtení dat, pokud je potřeba), RAS2 (aktivace precharge) a PRECHARGE (ukončení práce a deaktivace).

Samotná komponenta může vypadat například takto:

```
COMPONENT sdram
    PORT (
        -- system
        clk_133 : IN std_logic;
        reset : IN std_logic := '0';
```

```
        refresh : IN std_logic := '0';
        rw : IN std_logic := '0';
        we : IN std_logic := '0';
        addr : IN std_logic_vector(21 DOWNTO 0);
        data_i : IN std_logic_vector(15 DOWNTO 0);
        ub : IN std_logic;
        lb : IN std_logic;
        ready : OUT std_logic := '0';
        done : OUT std_logic := '0';
        data_o : OUT std_logic_vector(15 DOWNTO 0);

        -- SDRAM
        sdCke : OUT std_logic;
        sdCs : OUT std_logic;
        sdRas : OUT std_logic;
        sdCas : OUT std_logic;
        sdWe : OUT std_logic;
        sdBa : OUT std_logic_vector(1 DOWNTO 0);
        sdAddr : OUT std_logic_vector(11 DOWNTO 0);
        sdData : INOUT std_logic_vector(15 DOWNTO 0);
        sdUdqm : OUT std_logic;
        sdLdqm : OUT std_logic
        );
    END COMPONENT;
```

Vstupní kmitočet 133 MHz získáte například z PLL (ukázka je v kapitole o VGA). RESET slouží k úvodní inicializaci – spustí interní proces INIT. Refresh by měl vyvolávat hostitelský systém tak, aby jeho aktivace nekolidovala s požadavky na přístup k paměti. RW aktivuje paměť, WE povoluje zápis, adresa má 22 bitů (rozsah 4M slov), data 16 bitů. Datová sběrnice má oddělený vstup a výstup. Signály UB a LB maskují horní (upper) nebo dolní (lower) byte v datech, což je výhodné, když chcete zapsat například jen nižší byte a vyšší nechat beze změny (nastavíte UB na 1, LB necháte v 0).

Signál READY oznamuje, že paměť je připravena, tedy že proběhla inicializace. Protože je kontrolér spouštěn vždy inicializací paměti, můžeme tento výstup použít i jako RESET pro zbytek systému, aktivní v nule.

A konečně signál DONE oznamuje, že paměť má hotovo a systém může převzít data, případně poslat další požadavky.

Druhá část portu, tedy rozhraní pro fyzický čip SDRAM, pouze deklaruje všechny signály,

s nimiž se musí pracovat.

Samotná práce stavového automatu je řízena vnějšími požadavky a synchronizována náběžnou hranou hodin.

*Zdrojový kód je sice jednoduchý, ale rozsáhlý, a tak zájemce odkážu na web https://datacipy.cz, kde jsou všechny ukázkové kódy ke stažení.*

*Ukázka použití je v adresáři omdazz-alpha: počítač OMEN Alpha s procesorem T80 a 32 kB RAM – a jako RAM slouží právě externí čip SDRAM s řadičem. Použil jsem pro SDRAM frekvenci 100 MHz, aby byla dobře synchronizovatelná se zbytkem systému. PLL jsem nastavil tak, že se hodiny pro SDRAM fázově trochu předbíhají (posun o -50 °), takže se SDRAM lépe „trefuje" do cyklu procesoru.*

## 13.12 HDMI

HDMI je v současnosti de facto standard pro připojování video výstupů k zobrazovacím jednotkám (televizím, monitorům, projektorům). Jedná se o digitální rozhraní (na rozdíl od VGA, kde jednotlivé barevné složky jsou přenášené jako analogové signály), takže připojení přes HDMI by mělo být teoreticky ještě snazší než přes VGA.

Bohužel jen teoreticky – u HDMI narazíte na to, že jeho přenosové frekvence jsou ještě řádově vyšší než u VGA. Jestliže je například pixelová frekvence u určitého rozlišení VGA 25 MHz, potřebujete u HDMI přenášet 10 bitů na pixel, tedy pracovat s frekvencí 250 MHz. U jednočipů se tím dostáváme ve většině případů naprosto mimo jejich možnosti. Naštěstí u FPGA je třeba toto ještě frekvence, kterou zvládnou i starší a menší obvody.

*Pro velká rozlišení, 4K a podobné, potřebujete rozhodně modernější FPGA než ty, s nimiž pracujeme. Ty bývají vybavené i výkonnými serializéry, schopnými pracovat s gigahertzovými frekvencemi.*

Standardní HDMI konektor má 19 pinů. Část z nich slouží jako zemní vedení a stínění, je zde i napájení 5 V / 50 mA, je zde $I^2C$ sběrnice pro komunikaci s některými obvody v monitoru, ale pro nás je zajímavých 8 vodičů, které přenášejí 4 signály: TMDS Clock a TMDS Data 0 – TMDS Data 3.

*Signály jsou přenášeny po dvojicích vodičů, podobně jako u USB, Ethernetu a jiných vysokorychlostních sériových rozhraní. Jeden vodič se označuje jako plus, druhý jako minus (např. Clock+ a Clock-) a zjednodušeně lze říct, že by jejich úrovně měly být vzájemně inverzní. Pokud je Clock+ v log. 1, měl by být Clock- v log. 0.*

Možná vás zarazilo, že jsem zmiňoval přenos 10 bitů – HDMI přitom používá osm bitů pro určení intenzity každé barevné složky (R, G a B). Ve skutečnosti se těchto osm bitů převádí na desetibitový signál pomocí techniky, zvané *Transition-Minimized Differential Signalling* (odtud zkratka TMDS).

Toto kódování mění signál tak, aby se minimalizovalo vzájemné rušení ve vodičích a zvýšila tolerance k přeslechům u dlouhých nebo levných kabelů. Funguje, zhruba řečeno, podobně jako převod na Grayův kód, ovšem v závislosti na paritě volí jednu z dvou možných variant (XOR / XNOR), a to tu, která bude mít ve výsledku míň změn hodnoty. Informaci o tom, jaká varianta byla zvolena, přidá jako devátý bit. Poté podle počtu jedniček ve výsledku a podle „průběžné parity" případně datové bity neguje (a informaci o tom přidá jako desátý bit). TDMS zároveň vyhodnocuje „zatmění displeje", během kterého posílá specifické hodnoty.

*TMDS enkodér naleznete v příkladech ke knize na https://datacipy.cz*

Při generování obrazu můžeme postupovat úplně stejně jako u VGA, až do toho okamžiku, kdy máme digitální hodnotu barvových složek. U VGA displeje jsme je vyvedli ven a pomocí R-2R či jiného DAC převedli na analogovou hodnotu. Zde je pošleme do tří TMDS enkodérů a s rychlostí desetinásobku pixelové frekvence převedeme na 10 datových bitů. Ty serializujeme a od nejnižšího pošleme na jednotlivé TMDS Data (2 = Red, 1 = Green, 0 = Blue). Deset bitů dat přenášíme během jednoho pulsu TMDS Clock (ten tedy běží na pixelové frekvenci).

*S implementací serializátoru pomůže modul ALTLVDS (Altera), popřípadě OSERDES či OBUFDS (Xilinx)*

Z praktických důvodů se někdy používá frekvence nikoli desetinásobná, ale pětinásobná, a data jsou posílána při sestupné i vzestupné hraně.

Při vlastní práci se můžete odpíchnout od následujících odkazů.

Seznam rozlišení i s parametry:

https://www.mythtv.org/wiki/Modeline_Database

Implementace HDMI ve Verilogu:

https://github.com/charcole/NeoGeoHDMI

Implementace ve VHDL pro Spartan 7:

http://labs.domipheus.com/blog/hdmi-over-pmod-using-the-arty-spartan-7-fpga-board/

# 14 Vlastní mikroprocesor

## 14 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, věřte nebo ne, funkční mikroprocesor.

Ne že by neexistovaly doslova desítky podobných procesorů – viz kapitola o OpenCores. Existují a nazývají se „softcore microprocessors“. Některé z nich implementují reálné procesory (Z80, 6502, AVR apod.), jiné jsou naprosto proprietární a v podobě reálného čipu nikdy neexistovaly (například J1, kterému se budeme ještě věnovat).

Některé z těchto soft procesorů jsou velmi komplexní. Některé jsou optimalizované na rychlost, jiné na masivní paralelní práci, některé jsou postavené na reálném procesoru...

Procesor MHRD – říkejme mu tak – je velmi jednoduchý šestnáctibitový RISCový mikroprocesor. Nepoužívá instrukce v tom smyslu, v jakém jsme je poznali u ostatních mikroprocesorů. V podstatě má instrukce pouze dvě. 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 víc) 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 (tak to dělaly třeba procesory HD6809/HD6309 od Hitachi).

## 14.1 Architektura mikroprocesoru

Nechci tím zatěžovat 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 mysteriózní 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).*

*Bystré hlavy už jistě tuší, proč u procesoru 6502 je nutné před sčítáním vynulovat příznak přenosu a před odčítáním jej naopak nastavit...*

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". Používaly 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.

*Princip lze ilustrovat na 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, 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.

## 14.2 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.

Stejný postup je proveden i s hradlem XOR a jeho variantami XOR4B a XOR16B.

### Sčítačka

Po sestavení těchto základních kamenů přichází čas na oblíbené kousky: HALFADDER, FULLADDER a rozšíření: ADDER4B a ADDER16B. Už bych se opakoval, takže si zdrojový kód prosím vyhledejte v předchozím textu, nebo v příkladech ke knize.

### Multiplexor

Po sčítačce přichází na řadu multiplexor MUX – dva jednobitové vstupy, jednobitový výstup a jeden řídicí bit, který přepíná mezi vstupy.

Roztažením na 4 bity získáme komponentu MUX4B, dalším rozšířením pak MUX16B.

Ve hře MHRD je nejvýhodnější 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 <= in2 when (sel = '1') else in1;
end main;
```

Pokud přidáme další řídicí bit, získáme čtyřcestný šestnáctibitový multiplexor MUX4W16B.

Analogicky zkonstruujeme 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 zmiňovat. 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.

### Shifter, rotátor

Pro mikroprocesor MHRD nebude ani jeden z těchto elementů zapotřebí, ale hodí se je tu zmínit.

Shifter (či „barrel shifter“ - *český překlad „posouvač“ se mi opravdu nelíbí*) je obvod, který slouží k výpočtu logických či aritmetických posunů vlevo a vpravo. Staré osmibitové procesory mívaly instrukce rotací a posuvů, ale většinou jen o jednu pozici. Pokud jste chtěli například posunout hodnotu o tři pozice doleva (tedy vlastně „vynásobit osmi“), museli jste provést tři operace posunu po sobě.

Barrel shifter je obvod, který je ryze kombinační a dokáže vstupní hodnotu posunout o N bitů najednou. Například osmibitový shifter nabízí osmibitový vstup, osmibitový výstup a tři řídicí bity, které udávají, o kolik pozic se má vstup posunout (0-7).

Jako příklad vezměme posun vlevo. Klasická, „kanonická" implementace používá řetězec multiplexorů: M sloupců po N řádcích, kde N je počet bitů dat a M počet řídicích vodičů. Tedy u osmibitového multiplexoru je to 3 x 8 = 24 multiplexorů. Multiplexory v prvním stupni přepínají mezi bitem N a bitem N-1 (v nejnižším bitu nula). V druhém stupni se přepíná „ob dva", ve třetím „ob čtyři".

```
entity barrel is
  Port (
    A : in  STD_LOGIC_VECTOR (7 downto 0);
 S : in STD_LOGIC_VECTOR (2 downto 0);
    Q : out  STD_LOGIC_VECTOR (7 downto 0)
    );
end entity;

architecture mmux of Barrel is

signal I1:  STD_LOGIC_VECTOR (7 downto 0);
signal I2:  STD_LOGIC_VECTOR (7 downto 0);


begin

-- vstupy na I1

m1: entity work.mux port map (S(0),A(0),'0',I1(0));
mux1_loop: for i in 1 to 7 generate
mux_L1: entity work.mux port map (S(0),A(i),A(i-1),I1(i));
end generate;

-- I1 na I2

m2a: entity work.mux port map (S(1),I1(0),'0',I2(0));
m2b: entity work.mux port map (S(1),I1(1),'0',I2(1));
mux2_loop: for i in 2 to 7 generate
mux_L2: entity work.mux port map (S(1),I1(i),I1(i-2),I2(i));
end generate;
```

```
-- I2 na výstup

mux3_loop1: for i in 0 to 3 generate
mux_L3a: entity work.mux port map (S(2),I2(i),'0',Q(i));
end generate;


mux3_loop2: for i in 4 to 7 generate
mux_L3b: entity work.mux port map (S(2),I2(i),I2(i-4),Q(i));
end generate;


end architecture;
```

Nevýhoda tohoto řešení jsou dva mezistupně, I1 a I2. Při svém návrhu jsem si nejprve definoval osmicestný multiplexor (8-na-1) MUX8W:

```
entity mux8W is
   port ( sel : in  STD_LOGIC_VECTOR (2 downto 0);
            in0 : in  STD_LOGIC;
            in1 : in  STD_LOGIC;
            in2 : in  STD_LOGIC;
            in3 : in  STD_LOGIC;
            in4 : in  STD_LOGIC;
            in5 : in  STD_LOGIC;
            in6 : in  STD_LOGIC;
            in7 : in  STD_LOGIC;
            y: out STD_LOGIC
            );
end entity;

architecture main of mux8w is
begin
with sel select
    y <= in0 when "000",
 in1 when "001",
 in2 when "010",
 in3 when "011",
 in4 when "100",
 in5 when "101",
 in6 when "110",
```

```
 in7 when others;
end main;
```

Takových multiplexorů jsem následně použil osm, pro každý bit jeden.

```
architecture mux8L of barrel is

begin
mux0: entity work.mux8w port map (S,A(0),'0','0','0','0','0','0','0',Q(0));
mux1: entity work.mux8w port map (S,A(1),A(0),'0','0','0','0','0','0',Q(1));
mux2: entity work.mux8w port map (S,A(2),A(1),A(0),'0','0','0','0','0',Q(2));
mux3: entity work.mux8w port map (S,A(3),A(2),A(1),A(0),'0','0','0','0',Q(3));
mux4: entity work.mux8w port map (S,A(4),A(3),A(2),A(1),A(0),'0','0','0',Q(4));
mux5: entity work.mux8w port map (S,A(5),A(4),A(3),A(2),A(1),A(0),'0','0',Q(5));
mux6: entity work.mux8w port map (S,A(6),A(5),A(4),A(3),A(2),A(1),A(0),'0',Q(6));
mux7: entity work.mux8w port map (S,A(7),A(6),A(5),A(4),A(3),A(2),A(1),A(0),Q(7));

end architecture;
```

Pro mne je tento zápis čitelnější a „pochopitelnější“, hned vidím, co se děje a co se kam posouvá. Když budu chtít změnit shifter na „rotátor“, místo nul budu zase doplňovat bity A7..A1, když budu chtít udělat posun doprava, můžu místo nul doplňovat nejvyšší bit a získat tak aritmetický posuv...

Při testech vyšly obě řešení co do náročnosti na počet logických prvků totožně, syntetizér si s tím tedy poradil shodně. Je jen na vás, jakému zápisu dáte přednost.

## 14.3 Mikroprocesor MHRD

Mikroprocesor ve hře MHRD je šestnáctibitový procesor s Harvardskou architekturou a 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

Harvardská architektura znamená, že procesor má oddělené paměti a sběrnice pro data a pro instrukce. Podobný princip využívají často jednočipové mikropočítače. Oproti von Neumannově

architektuře mají tu výhodu, že mohou zároveň číst instrukci i pracovat s daty. Nevýhodou může být, že bez případných vnějších úprav nemohou provádět instrukce v operační paměti, tj. „chovat se k datům tak, jako by byly instrukce".

Mikroprocesor MHRD pracuje s šestnáctibitovými daty a šestnáctibitovým instrukčním slovem. S okolím komunikuje pomocí několika sběrnic:

| Sběrnice | Funkce |
|---|---|
| **instr** | Vstupní 16bitová sběrnice; hodnota z paměti programu na adrese, dané sběrnicí instrAddr |
| **instrAddr** | Výstupní 16bitová sběrnice; adresa instrukce. Obsahuje hodnotu registru PC |
| **dataAddr** | Výstupní 16bitová adresa pro práci s daty. Obsahuje hodnotu z registru M |
| **data** | Vstupní 16bitová sběrnice; hodnota z datové paměti na adrese, dané sběrnicí dataAddr |
| **result** | Výstupní 16bitová datová sběrnice. Spolu se signálem „write" slouží k zápisu do datové paměti |

### Součásti procesoru

V procesoru MHRD použijeme několik připravených komponent. Především dva šestnáctibitové registry REG16B v rolích registrů A a M. Dále pak šestnáctibitový čítač v roli registru PC a aritmeticko-logickou jednotku jako ALU.

Hlavním prvkem celého procesoru je dekodér, který rozloží instrukční slovo na základní komponenty a signály:

| Signál | Velikost (bity) | Význam |
|---|---|---|
| **constant** | 15 | Konstanta (15 / 5 bitů) |
| **cToM** | 1 | 1, pokud se má konstanta uložit do registru M |
| **loadA** | 1 | 1, pokud má být výsledek z ALU uložen do registru A |
| **loadM** | 1 | 1, pokud má být výsledek uložen do registru M |

| | | |
|---|---|---|
| **loadD** | 1 | 1, pokud se má výsledek zapsat do paměti (aktivuje výstup WRITE) |
| **opCode** | 4 | bity udávající operaci pro ALU |
| **jmpIfZ** | 1 | 1, pokud je nastavený příznak skoku |
| **op1** | 1 | První operand pro ALU (konstanta / A) |
| **op2** | 2 | Druhý operand (konstanta / A / M / paměť) |

Dekodér je prostý kombinační obvod s šestnáctibitovým vstupem a 27bitovým výstupem.

### Dekódování instrukcí MHRD

Instrukce procesoru MHRD nemají ve hře žádné mnemotechnické názvy a jsou popsány pouze pomocí popisu funkce jednotlivých bitů.

Pokud je bit 15 nastaven (=1), je zbytek instrukčního slova brán jako 15bitová konstanta, která je uložena do registru M.

Pokud je bit 15 nulový, je zbytek instrukčního slova rozdělen do několika částí se specifickým významem:

| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | d | d | a | b | b | o | o | o | o | j | c | c | c | c | c |

Bity „d“ (13-14) udávají cíl (destination) operace. Pokud je jejich hodnota 00, výsledek se zahodí. Pokud je to 01, výsledek se zapisuje do registru A. Pokud je jejich hodnota 10, výsledek se zapisuje do registru M. Pokud jsou rovny 11, bude výsledek zapsán do externí paměti dat na adresu, která je v registru M.

Bit „a“ (12) říká, co je první operand (A) pro aritmeticko-logickou jednotku ALU. Pokud je 0, bude prvním operandem obsah registru A, pokud je 1, bude to konstanta (bity c, viz níž).

Bity „b“ (10-11) udávají, co je druhý operand (B) ALU.

| B | Operand |
|---|---|
| **00** | Konstanta (bity c) |
| **01** | Obsah registru A |
| **02** | Obsah registru M |
| **03** | Data z datové sběrnice |

Bity „o“ (6-9) určují, jaká operace bude prováděna v ALU. Jsou přímo připojené na příslušné bity ALU a mají tento význam:

- Bit 6: Pokud je 1, je výstup negován.
- Bit 7: Pokud je 0, je na výstupu součet A a B, pokud je 1, je na výstupu výsledek funkce A AND B.
- Bit 8: Pokud je 1, je vstup B negován.
- Bit 9: Pokud je 1, je vstup A negován.

V praxi to znamená, že máme 16 možných operací:

| 9 | 8 | 7 | 6 | Operace |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | A + B |
| 0 | 0 | 0 | 1 | -1 - A - B |
| 0 | 0 | 1 | 0 | A AND B |
| 0 | 0 | 1 | 1 | A NAND B |
| 0 | 1 | 0 | 0 | A - B - 1 |
| 0 | 1 | 0 | 1 | B - A |
| 0 | 1 | 1 | 0 | A AND NOT B |
| 0 | 1 | 1 | 1 | A NAND (NOT B) |

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | B - A - 1 |
| 1 | 0 | 0 | 1 | A - B |
| 1 | 0 | 1 | 0 | NOT A AND B (implikace) |
| 1 | 0 | 1 | 1 | (NOT A) NAND B (negace implikace) |
| 1 | 1 | 0 | 0 | -2 - A - B |
| 1 | 1 | 0 | 1 | A + B + 1 |
| 1 | 1 | 1 | 0 | A OR B |
| 1 | 1 | 1 | 1 | A NOR B |

Kromě základních logických (AND, OR, NOR, NAND) a aritmetických (A+B, A-B, B-A) operací nabízí ALU i několik operací lehce podivných, které pravděpodobně nenajdou využití. Bohužel chybí bitový XOR.

Bit „j" je nazýván též jmpIfZ – „skok, pokud je výsledek 0". Technicky to znamená, že je-li výsledek z ALU roven nule, je do registru PC zkopírována hodnota z registru M.

Bity „c" (0-4) představují konstantu se znaménkem, to znamená, že při zpracování v ALU je bit 4 (nejvyšší bit konstanty) zkopírován do bitů 5-15.

### Dekodér

V dekodéru můžeme některé části operačního kódu přímo převést na výstupní signály – jde o bity a (=op1), b (=op2), o (=opCode) a j (=jmpIfZ).

Bity d je třeba převést na správné hodnoty výstupů loadA, loadM a loadD. Bity c musí být poslány na výstup „constant".

Pokud je nejvyšší bit = 1, je potřeba nastavit výstupy cToM a loadM.

### Zapojení procesoru

V procesoru musíme kromě registrů a dekodéru použít i několik multiplexorů:

- MUX16B pro výběr hodnoty, která bude zapsána do registru M (muxM)
- MUX16B pro výběr prvního operandu (mux1)

- MUX4W16B pro výběr druhého operandu (mux2)

Multiplexor muxM přepíná na základě signálu cToM mezi hodnotou „constant“ a výstupem ALU.

Multiplexor mux1 přepíná podle signálu op1 mezi hodnotou „constant“ (rozšířenou na 16 bitů) a výstupem registru A.

Analogicky mux2 přepíná mezi konstantou, registrem A, registrem M a vstupní sběrnicí *data*.

Do registrů A a M se zapisuje podle hodnot signálů loadM a loadA. Signál loadD můžeme spojit přímo s výstupem *write*.

Výstup čítače PC je zapojen na sběrnici *instrAddr*, výstup registru M na sběrnici *dataAddr*, výstup ALU na sběrnici *result*.

Vstup *reset* připojíme k nulovacímu vstupu čítače PC. Vstupní data pro čítač PC připojíme k výstupu registru MR a vstup *load* čítače PC vytvoříme jako logický součin (AND) signálu jmpIfZ a výstupu „zero“ z ALU.

A to je celé tajemství.

Každá instrukce procesoru MHRD trvá jeden takt hodin a zabírá v paměti jedno slovo. Vhodnou kombinací můžeme vytvořit několik skupin instrukcí, kterým můžeme dát mnemotechnické názvy pro snazší práci.

## Mnemotechnické názvy některých instrukcí

### LMI – Load M Implicit

LMI n, kde N je 15bitová hodnota. Bitově odpovídá hodnotě N s nejvyšším bitem rovným 1. Hodnota je zapsána do registru M.

### LDA – Load A

LDA přečte hodnotu z paměti na adrese v registru M a uloží ji do registru A. Musíme nastavit instrukci tak, aby operand 1 pro ALU byl roven 0 (bit a=1, bity c=00000), operand 2 je vstup z paměti (b=11), výsledek se ukládá do registru A (bity d = 01), operace může být třeba OR nebo sčítání (o=0000), bit j musí být nulový. Výsledný operační kód je tedy 0011 1100 0000 0000, tedy 3C00h.

**STA – Store A**

Podobně jako u LDA nastavíme součet s nulou (a=1, o=0000, c=00000), zdroj pro operand 2 bude registr A (b=01), cíl operace bude paměť (d=11). Bitově 0111 0100 0000 0000, tedy 7400h

**MMA – Move M to A**

Drobnou variací předchozích instrukcí změníme operand 2 a cíl tak, aby operand byl registr M a cíl registr A. Operační kód 0011 1000 0000 0000, 3800h.

**MAM – Move A to M**

Opět variace předchozího s kódem 0101 0100 0000 0000, 5400h

**LAI – Load A Implicit**

Instrukce LAI slouží k uložení konstanty v rozsahu -16..+15 do registru A. Vyjdeme z instrukce LDA, ale operand 2 bude konstanta (b = 00). Bitově 0011 0000 000c cccc, hexadecimálně 3000h až 301Fh.

**Aritmetické a logické instrukce**

Aritmetické operace mají jako jeden operand registr A. Druhý může být konstanta (označíme si ji I), registr M, registr A (dává smysl A+A pro posun doleva) nebo obsah paměti (symbolický název D). Cíl je registr M, registr A nebo paměť. Mnemotechnický zápis můžeme volit buď s plnými parametry (ADD A,M,A znamená A + M -> A), nebo vypustit první, protože je vždy A (ADD M,A), popřípadě zakomponovat cíl i zdroj do samotného názvu instrukce: ADDMA, ADDMM, ADDAD, ADDIA, ADDIM...

Možných kombinací instrukcí je mnoho, smysl dává označit si instrukce sčítání (ADD), odčítání (SUB), logických operací (AND, OR, NOT), inkrementace a dekrementace (INC, DEC), případně porovnání (CMP – když výsledek zahodíme, tedy nastavíme bity d=00). NOT můžeme realizovat např. jako NAND s konstantou 1Fh (znaménkově se rozšíří na FFFFh).

**JMP - Nepodmíněný skok**

Každá výše uvedená instrukce může zároveň sloužit jako instrukce skoku v případě, že výsledek operace v ALU je nulový. Cílová adresa je vždy v registru M. Pokud chceme provést nepodmíněný skok, musíme zajistit provedení „prázdné operace“ s výsledkem 0, tedy například nastavit operaci odčítání, jako oba operandy nastavit registr A (A-A bude vždy 0) a výsledek zahodit (d=00).

## Další vylepšení

S velmi podobným konceptem pracují Nisan a Schocken v knize „The Elements of Computing Systems“, resp. ve svém kurzu „From NAND to Tetris“, který, mimochodem, vřele doporučuju všem zájemcům o FPGA.

Kromě vylepšení ALU zvolili autoři o něco kompaktnější kódování operací. Místo krátkých konstant v těle instrukce použili několik bitů k implementaci osmi podmíněných skoků, které pracují nejen s příznakem „nula“, ale i s příznakem znaménka.

V knize dále celý koncept rozvíjejí – konstruují nad touto jednotkou další virtuální procesor, jehož instrukce jsou překládány na instrukce (spíše *mikroinstrukce*) tohoto procesoru. Virtuální procesor je schopen například pracovat se zásobníkem a volat podprogramy, což u procesoru MHRD přímo nelze.

Procesor MHRD nemá ani žádné instrukce pro rotace, posuny, nebo pro správu přerušení. Na druhou stranu je extrémně jednoduchý a „prostým okem pochopitelný“.

Můžete samozřejmě procesor „zesložitit“ a přidat do něj další obvody – třeba už popisovaný barrel shifter, zavést příznakové bity, nebo použít násobičku.

## Jak na CISC?

Základní princip výše uvedených procesorů (starší terminologií řečeno spíš *mikroprogramových řadičů*) spočívá v tom, že mají dlouhé instrukční slovo (zde 16 bitů) a jedna instrukce jim trvá jeden takt hodin. Dalo by se říct *téměř RISCové rysy*. Výhodou je rychlost a jednoduchost, nevýhodou to, že instrukce jsou poměrně „slabé“, v programech jich proto musíte použít víc, a program tak zabere víc paměti.

Jak to ale dělají CISCové procesory, že nabízejí komplexní instrukce, které se provádějí i několik taktů hodin?

*Upřesním: mám na mysli staré osmibitové procesory. U moderních strojů s prefetchem, s pipeline a s dalšími technickými vymoženostmi, včetně např. spekulativního provádění instrukcí, už dávno takové jednoduché pravdy neplatí.*

Ty úplně nejstarší měly vlastně velmi podobné vnitřní vybavení jako třeba MHRD – tedy ALU, registry, multiplexory, které nastavovaly cestu pro data, a kombinační logiku, která podle operačního kódu instrukce přenastavovala potřebné cesty. Hlavní rozdíl byl právě v mohutnosti tohoto dekodéru.

Procesor 6502, který je i díky projektu visual6502.org velmi dobře zdokumentovaný, měl dekodér, který se skládal ze dvou částí.

První se nazývala PLA, což bylo programovatelné logické pole (programmable logic array), které se chovalo jako asociativní paměť 130 slov po 21 bitech. Na vstup PLA bylo připojeno 21 bitů dat, složených z operačního kódu právě prováděné instrukce a z pořadového čísla instrukčního cyklu. Velmi zjednodušeně lze říct, že záznamy v této paměti vypadají například takto:

```
100XX100 X STY
```

Přeloženo: Pokud má instrukce operační kód, vyhovující masce „100XX100“, pak bez ohledu na číslo cyklu („X“) platí řádek s označením „STY“. Což sedí, protože odpovídají operační kódy 84h, 8Ch, 94h, 9Ch, což jsou různé varianty instrukce STY.

U každé instrukce může v jednom cyklu platit i více řádků. Informace o tom, jaké řádky v aktivní instrukci a aktuálním cyklu platí, vstupovala do druhé části dekodéru, do vlastní kombinační logiky, kde se z informací o platných řádcích skládaly informace pro multiplexory mezi jednotlivými akčními bloky.

Později mikroprocesory začaly používat mikroprogramové jádro, vlastně jakýsi „mikroprocesor v mikroprocesoru“, a překládat své instrukce na posloupnosti mikrooperací, které řídí jednotlivé části procesoru. Mikroprogramové jádro se totiž navrhne a odladí výrazně snáz než komplexní kombinační logika. Zkuste si třeba představit instrukční sadu procesoru Z80 a logiku, která určuje, co se má v kterém kroku kam poslat...

Moderní procesory používají mikroprogramování, a tak zatímco programátor běžně píše například instrukce pro architekturu x86, ve skutečnosti procesor vykonává vlastní mikrooperace, do kterých si instrukce x86 překládá.

# 15 Stručný úvod do Verilogu

# 15 Stručný úvod do Verilogu

Jazyk Verilog je podobný jazyku VHDL asi tak, jako je jazyk C podobný jazyku Pascal.

Tam, kde je Pascal velmi striktní a upovídaný, tam C používá různé zkratky a „rychlozápisy“ – místo begin a end složené závorky, přiřazení prostým rovnítkem namísto dvojznaku :=, nemusíte psát a=a+1, ale stačí a++ – ovšem na druhé straně se v C snadno ztratíte, překladač nic neřekne a vás čeká mnoho krásných chvil s hledáním chyby kvůli zapomenutému rovnítku...

Verilog působí na člověka, znalého VHDL, jako zjednodušený a zkondenzovaný zápis. Pojďme si připomenout naši oblíbenou neúplnou sčítačku ve VHDL:

```
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;
```

A teď stejný příklad ve Verilogu:

```
module halfAdder (A, B, Q,Cout);

  input  A;
  input  B;
  output Q;
  output Cout;

  assign Q   = A ^ B;  // operace XOR
  assign Cout = A & B;  // operace AND

endmodule // halfAdder
```

Na první pohled je vidět ten „céčkový“ přístup (pamětníci syntaxe podle pánů Kernighana a Ritchieho určitě poznají).

Na začátku se definuje modul, který zhruba odpovídá komponentě ve VHDL. Modul je deklarovaný podobně jako funkce v céčku, v závorkách jsou názvy signálů, které vstupují a vystupují. Následují deklarace těchto signálů – input pro vstupní, output pro výstupní. Poté už přichází na řadu přiřazení: výstupu Q se přiřazuje („assign“) výsledek operace A XOR B (a XOR je zapsáno stejným operátorem jako v C, tedy stříškou), výstupní přenos je A AND B, a pak je konec.

Určitě jste zaznamenali i komentáře, které začínají znaky //.

Podobnost jde ještě dál: Ve Verilogu, stejně jako v C, se rozlišují velká a malá písmena. Představte si, že se přepíšete a napíšete třeba toto:

```
assign q   = A ^ B;  // operace XOR
```

Překladač nezahlásí nic – tiše si vytvoří interní signál „q“, k výstupu Q nepřipojí nic, a vy problém zjistíte až při testu – výstup bude nedefinovaný.

Mimochodem, testování: Sice je možné míchat při syntéze moduly ve VHDL a Verilogu, ale ModelSim to neumí. Proto musíte mít i testbench napsaný ve Verilogu.

```
module half_adder_tb;

  reg bit1 = 0;
  reg bit2 = 0;
  wire w_SUM;
  wire w_CARRY;

  halfAdder uut
    (
     .A(bit1),
     .B(bit2),
     .Q(w_SUM),
     .Cout(w_CARRY)
     );

  initial
    begin
      bit1 = 1'b0;
      bit2 = 1'b0;
```

```
      #10;
      bit1 = 1'b0;
      bit2 = 1'b1;
      #10;
      bit1 = 1'b1;
      bit2 = 1'b0;
      #10;
      bit1 = 1'b1;
      bit2 = 1'b1;
      #10;
    end

endmodule // half_adder_tb
```

Testovací modul nemá žádné spojení se světem, a tak se seznam vstupů a výstupů vynechává. Klíčové slovo „reg“ definuje registr – entitu, která si zachovává svou hodnotu. Slovo „wire“ definuje vnitřní signál v modulu.

Následuje instance poloviční sčítačky – typ, název a přiřazení signálů, tedy obdoba ENTITY a PORT MAP z VHDL. Všimněte si „tečkové“ notace, která říká, ke kterému vstupu se jaký signál připojuje.

Poslední blok začíná slovem „initial“ – to označuje blok, který není syntetizovaný, je pouze pro simulaci, a jeho provádění začne na začátku simulace. Nastavují se hodnoty pro registry bit1 a bit2 a řádky „#10“ odpovídají něčemu jako „pause 10“ – pokračuje se tedy po zadané pauze.

Nastavování hodnot používá trochu nezvyklý zápis. Pojďme si jej vysvětlit.

## 15.1 Syntaktické základy Verilogu

**Komentáře** se zapisují podobně jako v C – buď na jeden řádek, uvozené znaky //, nebo na více řádků, uzavřené mezi /* a */.

**Operátory** jsou téměř stejné jako v C. Kromě bitových and, or, xor a not (to se neznačí vykřičníkem, ale vlnovkou: ~y je NOT Y) můžete použít i ternární operátor ?:

**Čísla** se zapisují podobně jako v C. Tedy např. 0x12 pro hexadecimální zápis apod. Verilog umožňuje zapsat hodnotu tak, že je ze zápisu patrné, kolik bitů zabírá. Používá se zápis:

```
[velikost]'[soustava][číslo]
```

Velikost je počet bitů, které hodnota zabírá (desítkově). Následuje písmeno, které značí základ číselné soustavy (B nebo b pro binární, O/o pro osmičkovou, D/d pro desítkovou, H/h pro šestnáctkovou). A následuje samotné číslo. Pár příkladů:

```
1'b1; //jeden bit s hodnotou 1
3'b100; //tři bity s hodnotou (binárně) 100
3'd4; //tři bity se stejnou hodnotou, zapsáno desítkově

16'hDEDA; //16bitová hodnota 0xDEDA
16'hDeda; //16bitová hodnota 0xDEDA - Verilog zde umožňuje míchat velká a malá písmena

32'hDeda_Baba; //32bitová hodnota 0xDEDABABA
```

Poslední příklad ukazuje syntaktický cukr, který Verilog nabízí – pro lepší čtení můžete čísla po šestnácti bitech oddělovat podtržítkem.

Od verze Verilog 2001 můžete zadávat i hodnoty se znaménkem, a to pomocí prefixu „s“:

```
3'sd4; //tři bity s hodnotou 101, ale se znaménkem
```

Informace o tom, že hodnota je se znaménkem, se použije například při operaci bitového posunu nebo při rozšiřování na více bitů.

Pro převod mezi znaménkovými a neznaménkovými čísly lze použít konverzní funkce *$signed()* a *$unsigned()*.

Bez velikosti se vezme nejmenší dostačující počet bitů. Bez určení soustavy se uvažuje desítková.

**Řetězce** se píšou do uvozovek.

**Hodnoty** logických signálů jsou 0, 1, x (pro neznámou hodnotu) a z (vysoká impedance, odpojený stav).

## 15.2 Datové typy

**Signál**, známý z VHDL, se ve Verilogu označuje slovem „wire“ – tedy česky „drát, vodič“. Vodiče (wires) se používají k propojování logických elementů, stejně jako v reálném světě.

```
wire carry;
```

Podobně jako ve VHDL můžeme spojit několik vodičů, co spolu logicky souvisejí, do tzv. **sběrnice**. Zápis je podobný VHDL, ale je opět zkondenzovaný:

```
wire [7:0] data;
```

Právě jsme definovali sběrnici (vektor) „data". Jednotlivé vodiče můžeme odkazovat zápisem data[0], data[1] atd., popřípadě i na více vodičů naráz: data[3:0]. Pozor na to, abyste při částečném výběru zachovali logiku posloupnosti, tj. buď klesání, nebo stoupání. Pokud vektor definujete jako [7:0], je v pořádku z něho vyjmout část [5:2], ale při pokusu vyjmout [2:5] dojde k chybě.

**Proměnné**, které ve VHDL vlastně neexistují (mimo procedury), jsou ve Verilogu přítomny a definují se podobně jako signály (wire), jen místo slova „wire" používáme slovo „reg":

```
reg clk;
reg [7:0] buffer;
```

Pokud tušíte souvislost se slovem „registr", jste na správné stopě. V jazyce Verilog jde opravdu o abstraktní vyjádření prvku, který udržuje svoji hodnotu (např. jako klopný obvod).

Datový typ **integer** je všeobecně použitelný 32bitový registr, na jehož hodnotu se hledí jako na celé číslo. Podobně typ **real** slouží k uložení reálných čísel (typicky 64bitový IEEE formát).

Pro práci s časem existují typy **time** a **realtime**, zde se jim nebudu věnovat.

**Řetězec** lze přiřadit vektoru, který má dostatečnou šířku v bitech (8 x počet znaků). Pokud má znaků víc, přebývající se nepoužijí, pokud míň, je vektor doplněn zleva nulami.

```
reg [8*5:1] str = "Hello";
```

**Pole** (array) slouží k deklaraci více prvků stejného typu, které mají logickou souvislost. Syntax je opět velmi podobná céčku:

```
integer a[0:9]; // pole s deseti prvky typu integer
reg[7:0] memory[0:1023]; // 1 kB RAM (1024 x 8 bitů)
wire[7:0] video[1:3][479:0][319:0]; // trojrozměrné pole
```

Pole nemůžeme, na rozdíl od vektorů, přiřadit jako celek, musíme se odkazovat na konkrétní prvek.

## 15.3 Operátory

Verilog nabízí celou sadu operátorů, logických, aritmetických i relačních. Nemá smysl je probírat do hloubky, prostý výpis postačí:

| Operátory | Význam |
|---|---|
| **+, -, *, /** | Sčítání, odčítání, násobení, dělení |
| **%** | Zbytek po dělení (modulo) |
| ****** | Umocňování (a**b = $a^b$) |
| **<, >, <=, >=** | Porovnávání: menší, větší, menší nebo rovno, větší nebo rovno |
| **===, !==** | Rovná se, nerovná se (porovnává i hodnoty X a Z) |
| **==, !=** | Rovná se / nerovná se. Pokud je některý z operandů X nebo Z, výsledek je X |
| **&&, \|\|, !** | Logické and, or, not |
| **&, \|, ^, ~** | Bitové and, or, xor, not |
| **<<, >>** | Logický posun vlevo, vpravo |
| **<<<, >>>** | Aritmetický posun vlevo, vpravo |

## 15.4 Moduly

S moduly jsme se už setkali na začátku kapitoly. Připomenu, že modul je ve Verilogu blok kódu, který popisuje nějakou funkcionalitu. Použití má podobné jako komponenta ve VHDL. Modul má vždy nějaké jméno a jeho definice může obsahovat případně *port*, tedy tu část, která komunikuje s okolním světem.

```
module <jméno> ([port]);
  // Obsah modulu
endmodule

// Modul bez portu
module jmeno;
  // Obsah modulu
endmodule
```

Jméno modulu slouží jako "název typu" – když tento modul používáte v jiném modulu, deklarujete instanci právě tímto jménem. Vraťme se na chviličku k naší poloviční sčítačce, jak jsme si ji definovali na začátku kapitoly, a pomocí dvou polovičních si složíme jednu celou:

```
module fullAdder (A, B, Cin, Q, Cout);

  input  A;
  input  B;
  input  Cin;
  output Q;
  output Cout;

  wire sub;
  wire c1,c2;

  halfAdder first (.A(A),.B(B),.Q(sub),.Cout(c1));
  halfAdder second (.A(sub),.B(Cin),.Q(Q),.Cout(c2));

  assign Cout = c1 | c2;  // operace OR

endmodule // full_adder
```

Všimněte si, že instance modulu (zde *first* a *second*) jsou pojmenované, i když se na ta jména nikde neodkazujeme. Stejně jako u plné sčítačky ve VHDL jsme i zde použili dvě poloviční, vodič pro mezisoučet (sub), dva vodiče pro přenosy z jednotlivých sčítaček a jedno hradlo OR.

Samozřejmě můžete vytvářet moduly z modulů a tyto moduly používat v jiných modulech. Ve Verilogu, stejně jako ve VHDL, existuje pojem *top-level module*, neboli modul nejvyšší úrovně. Je to takový modul, který obsahuje ostatní moduly a sám není v žádném jiném obsažen. Kupříkladu testovací modul (testbench) je top-level modulem – obsahuje ostatní, ale sám není obsažen.

V hierarchii modulů se můžeme na vnitřní moduly odkazovat pomocí tečkové notace – např. „full_adder.first“ odkazuje na instanci „first“ poloviční sčítačky, která je obsažena v plné sčítačce atd.

## 15.5 Porty

S porty, tedy vstupními a výstupními vodiči, jsme se už setkali. Podobně jako u VHDL, i ve Verilogu má každý port určený směr: vstupní (input), výstupní (output) nebo obousměrný (inout). Plná definice portu má tvar:

```
směr [typ] [rozsah] jméno [,jméno...]
```

Směr je už zmíněný input, output, nebo inout.

Typ je „wire“, pokud není zadáno jinak. Pro vstupy nebo vstupně-výstupní signály nemá smysl jiný typ než „wire“, ale u výstupního signálu můžeme zadat typ „reg“ a předepsat tak, že výstup udržuje svou hodnotu, dokud není uvnitř obvodu změněna.

Rozsah použijete v případě sběrnic. Poslední součást deklarace portu je povinné jméno, případně jména, pokud má mít modul více portů stejného typu.

V původním Verilogu se port definoval tak, jak jsme si ukázali – součástí port listu byly pouze názvy, a jejich typy se určovaly až za hlavičkou modulu. A, opět analogicky s jazykem C, i ve Verilogu časem přesunuli typy portů do deklarace modulu, takže naše plná sčítačka může mít od verze Verilog 2001 stručnější zápis:

```
module fullAdder (input A, B, Cin, output Q, Cout);
```

Všimněte si, že jsem využil možnost zapsat víc signálů stejného typu (zde „input wire“) v jedné deklaraci.

Při použití modulu můžete, podobně jako ve VHDL, zapsat přiřazení signálů v takovém pořadí, v jakém jsou zapsané v definici modulu:

```
halfAdder first (A, B, sub, c1);
```

Někdy ale může být pohodlnější využít zápis s tečkovou notací:

```
halfAdder first (.A(A),.B(B),.Q(sub),.Cout(c1));
// .PORT(signál)
```

Tečka a název portu určuje, ke kterému portu se připojí signál (vodič nebo registr), jehož jméno je v závorce. Tuto notaci můžeme využít i k tomu, abychom explicitně zapsali nepřipojený port: .Q() říká, že port Q není k ničemu připojen. Pokud je to vstup, má nedefinovanou hodnotu „z“ (vysoká impedance).

## 15.6 Příkaz assign

Ve Verilogu je assign základní příkaz, který *přiřazuje* vodiči nebo sběrnici nějaké propojení. Příklady jsme už viděli:

```
  assign Q   = A ^ B;  // operace XOR
  assign Cout = A & B;  // operace AND
  assign Cout = c1 | c2;  // operace OR
```

Podobně lze přiřazovat i celé sběrnice:

```
wire [7:0] data1, data2;
wire [15:0] adresa;

assign data1 = data2; //propojuje dvě sběrnice

assign adresa[15:8] = data1; //vyšších 8 bitů adresy bude data1

//složené závorky označují spojení dvou sběrnic
assign adresa = {data1, data2};

// i na levé straně
assign {data1, data2} = adresa;

assign data2 = {data1[3:0], data1[7:4]}; //prohození vyšší a nižší poloviny slova
```

Příkaz „assign“ si můžeme představit jako návod k připojování vodičů: „tento vodič připojte k výstupu hradla NAND...“

Ve Verilogu můžete použít i implicitní přiřazení v okamžiku, kdy deklarujete vodič. Místo konstrukce

```
wire z;
assign z = a & b;
```

můžete zapsat

```
wire z = a & b;
```

S příkazem *assign* si vystačíte pro všechny případy kombinační logiky.

### Kombinační obvody ve Verilogu

#### Plná sčítačka

```
module fullAdder ( input a, b, cin,
            output sum, cout);

  assign sum  = (a ^ b) ^ cin;
  assign cout = (a & b) | ((a ^ b) & cin);
endmodule
```

#### Multiplexor 2-na-1

```
module mux (input a, b, sel,
            output c);

  assign c = sel ? a : b;
endmodule
```

#### Dekodér 3x8

```
module dec_3x8 (input       en,
                input [2:0] in,
                output[7:0] out);

  assign out = en ? 1 << in : 0;
endmodule
```

K výstupu *out* je zde přiřazena hodnota „1“, posunutá o tolik bitů doleva, kolik udává vstup *in*, ovšem pouze v případě, že povolovací vstup *en* je roven 1.

## 15.7 Blok always

Opět něco, co připomíná programování. Stejně jako jsou ve VHDL k dispozici procedury, jsou ve Verilogu k dispozici procedurální bloky. Jeden z nich jsme si už ukázali – byl to blok „initial“, který jsme použili v testovacím modulu.

Procedurální blok má, podobně jako u VHDL, tu vlastnost, že se provádí sekvenčně. *Ve skutečnosti jej syntezátor přepočítá do paralelní podoby, ale kód vypadá jako zápis algoritmu...*

Blok *always* se provádí vždy, když dojde k nějaké události, tedy většinou při změně hodnoty některého signálu. Stejně jako u VHDL i zde se definuje tzv. *sensitivity list*, tedy seznam signálů, na které je daný blok citlivý a jejichž změnu hlídá.

```
always @ (sensitivity list) begin
// ...
end
```

Tedy například: *always @ (a or b)* se vykoná vždy, když se změní signál a nebo b.

Nejčastější případ senzitivity je „posedge clk" – tedy „blok se vyvolá při vzestupné hraně (positive edge) signálu clk". Existuje i zrcadlová varianta „negedge"

```
always @ (posedge clk) begin
// ...
end
```

*Pokud blok nemá zadanou žádnou citlivost, stane se to, že při simulaci běží stále. Funguje jako nekonečná smyčka, která úspěšně zasekne celou simulaci! Přesto je možné i takový blok použít smysluplně, pokud mu dáme nějaké rozumné zpoždění, např. „always #10 clk = ~clk;" bude generovat hodinový signál.*

Signály, které jsou v bloku *always* použité, do nichž se přiřazuje, by měly být vždy typu „proměnná" – tedy vlastně „reg" (registr).

Blok *always* můžete použít i při definování kombinačních obvodů. Co byste řekli na takovouto sčítačku?

```
module fullAdder (input a, b, cin,
      output reg sum, cout);

  always @ (a or b or cin) begin
    {cout, sum} = a + b + cin;
  end

endmodule
```

Ale hlavní použití najde při konstrukci sekvenčních obvodů – klopných obvodů, čítačů, registrů, pamětí a dalších, kde vstupuje do hry časový synchronizační signál.

Pro ukázku si nadefinujme pár sekvenčních obvodů.

## Sekvenční obvody ve Verilogu

### Klopný obvod D s asynchronním signálem reset

```
module dff ( input d, rst, clk,
             output reg q);

  always @ (posedge clk or posedge rst)
       if (rst)
          q <= 0;
       else
          q <= d;
endmodule
```

Potřebujeme, aby obvod reagoval jak na signál hodin, tak na signál reset, proto jsou oba signály v sensitivity listu. U reset je rovněž specifikace *posedge*, protože nás nezajímá případ, když signál reset přechází do nuly (tedy končí).

## Klopný obvod D se synchronním vstupem reset

```
module dff ( input d, rst, clk,
             output reg q);

  always @ (posedge clk)
       if (rst)
          q <= 0;
       else
          q <= d;
endmodule
```

Zde se blok always neohlíží na případné změny signálu reset, takže klopný obvod je vynulován pouze tehdy, když je signál reset aktivní v okamžiku vzestupné hrany hodinového signálu.

## Čítač modulo 10

```
module counter_10 (input clk,
                   input rst,
                   output reg[3:0] out);
```

```
  always @ (posedge clk) begin
    if (rst) begin
      out <= 0;
    end else begin
      if (out == 10)
        out <= 0;
      else
        out <= out + 1;
    end
  end
endmodule
```

## 15.8 Testování – blok initial

Už jsme si ukazovali jeden jednoduchý *testbench*. Ve Verilogu máme k dispozici speciální blok „initial“, v němž můžeme zapisovat i konstrukce, které se mění s časem (tedy nesyntetizovatelné). Pojďme se podívat na zjednodušený test poloviční sčítačky:

```
module halfAdder_tb;

  reg bit1 = 0;
  reg bit2 = 0;
  wire w_SUM;
  wire w_CARRY;

  halfAdder uut
    (
     .A(bit1),
     .B(bit2),
     .Q(w_SUM),
     .Cout(w_CARRY)
     );

  initial
    begin
      bit1 = 1'b0;
      bit2 = 1'b0;
      #10;
      $display ("%0b + %0b = %0b%0b",bit1,bit2,w_CARRY, w_SUM);
      bit1 = 1'b0;
```

```
      bit2 = 1'b1;
      #10;
      $display ("%0b + %0b = %0b%0b",bit1,bit2,w_CARRY, w_SUM);
      bit1 = 1'b1;
      bit2 = 1'b0;
      #10;
      $display ("%0b + %0b = %0b%0b",bit1,bit2,w_CARRY, w_SUM);
      bit1 = 1'b1;
      bit2 = 1'b1;
      #10;
      $display ("%0b + %0b = %0b%0b",bit1,bit2,w_CARRY, w_SUM);


    end

endmodule // half_adder_tb
```

Speciální funkce $display se postará o výpis zadaných hodnot (podobně jako printf() v jazyce C). I tento zápis můžeme zjednodušit:

```
module halfAdder_tb;

  reg bit1 = 0;
  reg bit2 = 0;
  wire w_SUM, w_CARRY;
  integer i;

  halfAdder uut
    (
     .A(bit1),
     .B(bit2),
     .Q(w_SUM),
     .Cout(w_CARRY)
     );

  initial
    begin
      bit1 = 0;
      bit2 = 0;
      bit3 = 0;
```

```
$monitor("%0b + %0b + %0b = %0b%0b",bit1,bit2, bit3, w_CARRY, w_SUM);

      for (i = 0; i < 8; i = i + 1) begin
      {bit1,bit2,bit3} = i;
         #10;
end
    end

endmodule // half_adder_tb
```

Funkce $monitor() funguje stejně jako display, ale s tím rozdílem, že se vyvolá vždy, když se některý z jejích parametrů změní.

Všimněte si i použití cyklu for() a následného přiřazení bitů.

Bloků initial může být libovolný počet. Při simulaci se spustí všechny naráz.

```
initial begin
    a = 5;
#20 b = 10;
end

initial begin
#10 a = 10;
#40 b = 5;
end

initial begin
#60 $finish;
end
```

Všechny tři bloku se spustí v čase t=0. Proměnné a bude přiřazena hodnota 5. Po 10 ns se přiřadí do a hodnota 10 (druhý blok). Po dalších 10 ns (tedy v čase t=20) proběhne příkaz b=10. V čase t=50 se provede b=5 (tedy 40 ns po přiřazení a=10). V čase t=60 se projeví třetí blok a provede speciální úlohu *$finish*, která ukončí simulaci.

V bloku *initial* jsou prováděny příkazy tak, jak jsou zapsány za sebou mezi slovy *begin* a *end*. Lze ale i v rámci jednoho bloku vynutit paralelní provádění:

```
initial begin
  #10 a = 5;
  fork
#20 a = 10;
#10 a = 20;
  join
end
```

Příkazy mezi slovy **fork** a **join** se provedou paralelně. V tomto případě tedy 10 ns po začátku simulace bude a=5, a pak se spustí dva příkazy naráz. Jeden po 10 ns od spuštění (tedy v čase t=20) změní a na 20, druhý po 10 ns od spuštění (v čase t=30) změní a na 10.

Určitě jste si všimli, že v bloku *initial* přiřazujeme pomocí symbolu „=“. To je proto, že zde opravdu přiřazujeme hodnotu do proměnné (registru). Není to přiřazení (assign), které u vodiče říká, co je kam připojené.

Procedurální přiřazení pomocí „=“ se nazývá *blokující přiřazení*. To proto, že je přiřazena hodnota okamžitě. Podobně jako ve VHDL, když se v proceduře přiřazuje do proměnné.

Existuje i varianta neblokujícího přiřazení „<=“ – takové přiřazení se „poznamená“, ale hodnota bude změněna až na konci „časového kroku“.

## 15.9 Stručné shrnutí základů Verilogu

Ve Verilogu je stavební jednotka „modul“. Vše, co definujete, zapisujete mezi **module** a **endmodule**. Modul může (ale nemusí) mít definované **porty**, tedy vstupy a výstupy, kterými komunikuje s okolím.

Moduly můžete používat v jiných modulech.

Dva základní typy signálů jsou **wire** a **reg**. Registr reg můžeme považovat za svého druhu *proměnnou*, tedy objekt, který si pamatuje přiřazenou hodnotu. Vodič **wire** je analogií elektrického vodiče a slouží jen k propojení portů v modulech.

Ve Verilogu jsou tři základní typy bloků: **always**, který se provádí vždy, když dojde ke změně některých signálů. **Initial**, který se používá hlavně při simulaci a spouští se na začátku simulace. Třetí blok (přesněji příkaz) je **assign**, kde probíhá přiřazení.

Bloky **always** a **initial** umožňují provést i více příkazů naráz. Stačí je zabalit do konstrukce **begin ... end;**

Pamatujte si, že:

- do registru (reg) lze přiřadit pouze v bloku **initial** nebo **always**
- vodiči (wire) přiřazujeme hodnotu pouze pomocí **assign**.

## 15.10 Parametrizace modulů

Blok **generate** uspoří spoustu psaní velmi podobných příkazů. Pokud si vzpomínáte, jak jsme ve VHDL dělali z poloviční sčítačky vícebitovou, můžeme si to zkusit i ve Verilogu:

```
module genericAdder
  #(parameter N=4)
    (  input [N-1:0] a, b,
      output [N-1:0] sum, cout);

  // Deklarujeme proměnnou, která se použije pouze
  // pro generování

  genvar i;

  // Vygenerujeme si čtyři poloviční sčítačky
  generate
    for (i = 0; i < N; i = i + 1) begin
          halfAdder u0 (a[i], b[i], sum[i], cout[i]);
    end
  endgenerate
endmodule
```

V tomto modulu používáme i parametr – všimněte si konstrukce #() před samotným „port listem“, včetně zápisu defaultní hodnoty.

Takový modul použijeme stejně jako běžný modul, ale nesmíme zapomenout uvést parametry:

```
názevModulu #(parametry) názevInstance (port map);
```

Tedy například:

```
genericAdder #(4) adder (a, b, sum, cout);
```

Čtyřbitová sčítačka z polovičních sčítaček nemá moc smysl, udělejme si tedy reálnou čtyřbitovou. Přidáme vnitřní přenos a port cin. Port cout bude jen jednobitový:

```
module genericAdder
  #(parameter N=4)
    (  input [N-1:0] a, b,
     input cin,
      output [N-1:0] sum,
    output cout);

  wire[N:0] carry; // O 1 bit navíc
  // - poslouží přenos z nejvyššího řádu

  // Deklarujeme proměnnou, která se použije pouze
  // pro generování

  genvar i;

  // Vygenerujeme si čtyři plné sčítačky
  generate
    for (i = 0; i < N; i = i + 1) begin
      fullAdder u0 (a[i], b[i], carry[i], sum[i], carry[i+1]);
    end
  endgenerate

  // přenos pro nejnižší bit přichází zvenčí,
  // přenos z nejvyššího bitu vystupuje ven
  assign cout = carry[N];
  assign carry[0] = cin;
endmodule
```

V bloku *generate* můžeme kromě smyčky FOR použít i konstrukci *if ... else*, popřípadě ekvivalent přepínače switch – ve Verilogu se jmenuje *case*:

```
module myAdder (input a, b, cin,
                output sum, cout);
  parameter ADDER_TYPE = 1;

  generate
    case(ADDER_TYPE)
      0 : halfAdder u0 (.a(a), .b(b), .sum(sum), .cout(cout));
```

```
      1 : fullAdder u1 (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
    endcase
  endgenerate
endmodule
```

Můžeme si nastavit parametr, který určí, jakou sčítačku nakonec syntetizér použije. Klíčové slovo *parameter* můžete vnímat jako definici pojmenované konstanty.

## 15.11 Blokové instrukce

V blocích (always, initial apod.) můžeme použít základní řídicí instrukce, jako je if-else nebo smyčky. Platí, že tělo je vždy jedna instrukce. Pokud je potřeba víc instrukcí, můžete je uzavřít do bloku begin...end.

```
// if bez else
if (výraz)
  [příkaz]

// if - else
if (výraz)
  [příkaz]
else
  [příkaz]

// if - else a složený příkaz
if (výraz) begin
  [příkazy]
end else begin
  [příkazy]
end

// if-else-if
if (výraz)
  [příkaz]
else if (výraz)
  [příkaz]
else
  [příkaz]
```

Nekonečná smyčka:

```
forever [příkaz]
```

Smyčka s počtem opakování:

```
repeat (opakování) begin
  [příkazy]
end

repeat (opakování) @ (událost) begin
  [příkazy]
end
```

Cyklus s podmínkou – provádí se, dokud je pravdivá:

```
while (podmínka) begin
  [příkazy]
end
```

Cyklus for – s tím jsme se už setkali, má stejnou syntax jako v jazyce C:

```
for (inicializace; podmínka; změna hodnoty) begin
  [příkazy]
end
```

Přepínač case – podle hodnoty výrazu se volí operace:

```
case (výraz)
  hodnota1 :   příkaz
  hodnota2,
  hodnota3 :   příkaz
  hodnota4 :   begin
                  příkazy
               end
  default  :  příkaz
endcase
```

Není potřeba v jednotlivých větvích používat *break*, jako je tomu v jazyce C. Konstrukce *case* je podobnější stejnojmenné konstrukci z jazyka Pascal.

## 15.12 A dál?

Stručný úvod do Verilogu rozhodně nepokrývá všechny možnosti tohoto jazyka. Jeho největší síla je v podobnosti s jazykem C, který je v oblasti počítačů a elektroniky v současnosti de facto *lingua franca*, a v možnostech behaviorálního modelování a popisu testů pro simulace. Pro člověka, který přichází do světa FPGA s nějakými znalostmi z programování, bude Verilog, respektive jeho následovník SystemVerilog, možná snazším způsobem, jak se do problematiky dostat.

https://www.chipverify.com/verilog/verilog-tutorial
https://www.hdlworks.com/hdl_corner/verilog_ref/

Trendu „programování logiky" jde ještě víc naproti jazyk SystemC – nadstavba k jazyku C++, kdy požadované funkce namodelujete prostředky z jazyka C++, tedy včetně tříd a objektů. Výhodou takového přístupu je, že vývojáře znalého programování v C++ naleznete snáz než specialistu na VHDL / Verilog. Navíc pro C++ existuje velké množství vývojových prostředí a dalších nástrojů.

# 16 Verilog prakticky

# 16 Verilog prakticky

Přemýšlel jsem, na čem demonstrovat menší projekt ve Verilogu a jeho případné spojení s VHDL, a nakonec jsem si říkal, že nebudu chodit daleko a použiju příklad, který jsem už v textu zmínil, totiž Gameduino.

Jen pro připomenutí: jedná se o shield k Arduinu, který ve FPGA Spartan 3 implementuje některé funkce grafické karty a nabízí výstup na VGA a stereo zvuk. Jeho autor, James Bowman, implementoval i jednoduchý grafický koprocesor, a postavil jej nad svým vlastním procesorem J1. A právě tento procesor může být hezkou ukázkou popisu obvodů ve Verilogu.

## 16.1 FORTH a procesor J1

Procesor J1 je *zásobníkový mikroprocesor*, který implementuje minimální sadu instrukcí pro jazyk FORTH.

*Jazyk FORTH – dovolte odbočku – si vytvořil Charles „Chuck" Moore na konci 60. let, a to jako vývojářskou pomůcku pro svou práci, totiž programování systémů na řízení radioteleskopů. Byla to jeho pomůcka, a myslím, že se dá směle říct, že nese dodnes velmi silný vtisk jeho osobnosti. Zatímco ostatní jazyky se snaží přiblížit počítač programátorovi, FORTH jako by šel opačnou cestou: je velmi jednoduchý pro počítač, ale náročný pro programátora. Dokonce se tvrdí, že některá slova tohoto jazyka Moore vymýšlel tak, aby se mu pohodlně psaly na klávesnici... V Československu se FORTH dostal do povědomí veřejnosti v osmdesátých letech, kdy o něm začal vycházet seriál v časopise Amatérské Radio. Říkalo se, že nějaká ústřední plánovací komise rozhodla, že FORTH bude ten správný jazyk pro budoucí socialistické mikropočítače, ale pak to s tím socialismem nějak nedopadlo...*

Hlavním rysem jazyka FORTH je, že používá dvojici zásobníků (uživatelský zásobník a zásobník návratových adres). Na zásobníku očekávají procedury (ve FORTHu se nazývají „slova") své parametry a nechávají tam výsledky své činnosti.

Tato podstata vede k zajímavému důsledku: FORTH nezapisuje aritmetické operace tak, jak jsme zvyklí, s operátory mezi operandy („a + 1" – *infixová notace*), ale v notaci *postfixové* („a 1 +").

*Opačná, prefixová notace, kdy je zapsán nejdřív operátor a pak operandy („+ a 1"), může být povědomá těm, kdo znají jazyk LISP. Navrhl ji polský logik Jan Łukasiewicz kolem roku 1920, a proto je známa i jako polská notace. Její obrácená varianta, postfixová, je tedy logicky nazývána „obrácenou polskou notací", v angličtině Reverse Polish Notation (RPN).*

Obrácená polská notace lehce znepřehledňuje zápis aritmetických operací („2+3*4“ zapíšete jako „3 4 * 2 +“, popřípadě jako „2 3 4 * +“), na druhou stranu výrazně zjednodušuje zpracování: interpreter nemusí řešit závorky ani prioritu či asociativitu operátorů. Operátor „+“ prostě vezme dvě hodnoty ze zásobníku, sečte je a výsledek zase uloží na zásobník. Operátor násobení „*“ udělá totéž, ale hodnoty vynásobí. Je na programátorovi, aby zohlednil priority.

Největší síla a důvod, proč se FORTH rozšířil a nezůstal soukromou pomůckou geniálního, i když možná lehce šíleného vývojáře, je v tom, že si ve FORTHu může každý vytvářet vlastní příkazy („slova“). Respektive ještě jinak: Když programujete ve FORTHu, vytváříte si tím vlastní dialekt FORTHu, plný krátkých a efektivních „slov“, která se skládají do složitějších a složitějších, až nakonec na nejvyšší úrovni zůstane jedno či několik mocných slov, která udělají všechno to, co potřebujete („ŘIĎ-TELESKOP!“). Zkrátka jazyk, strukturovaný z podstaty, v době, kdy „strukturované programování“ bylo předmětem vášnivých akademických diskusí.

Díky těmto rysům je FORTH velmi efektivní, rychlý a implementačně jednoduchý. Nerozšířil se jako univerzální jazyk pro vývoj software, ale je samozřejmě dostupný pro všechny možné operační systémy a své použití nachází často právě ve světě zabudované („embedded“) elektroniky.

Chuck Moore navrhl i vlastní mikroprocesor, který by dokázal přímo „podporovat FORTH“ – jmenoval se Novix N4000. Později jej vylepšil a koncepci prodal společnosti Harris. Ta jej vylepšila na verzi RTX2000, přidala radiační ochranu a našla pro něj zákazníka: vesmírný program. Dnes společnost Intersil vyrábí verzi RTX2010, který například řídil přistávací modul Philae při misi Rosetta.

Bowmanův procesor J1 je mnohem jednodušší (200 řádků Verilogu) než tyto specializované procesory, ale přesto překvapivě efektivní a rychlý. V dokumentaci autor uvádí, že s tímto procesorem streamoval nekomprimované video přes Ethernet jen pomocí jednoduché programové smyčky.

J1 je šestnáctibitový dvouzásobníkový procesor s harvardskou architekturou a instrukční sadou, která je velmi blízká standardu ANS FORTH. J1 neimplementuje některé funkce: příznak přenosu, přerušení, výjimky, relativní skoky, násobení a dělení nebo osmibitový přístup k paměti. I s těmito omezeními jde o výkonný procesor, schopný mnoha komplexních úloh (a v roli grafického koprocesoru ideální).

Obsahuje zásobník návratových adres RS (16 položek) a uživatelský datový zásobník DS (17 položek). Nejvyšší položka zásobníku RS je označována R. Nejvyšší položka datového je T (TOS, Top Of Stack), položka pod ní je N (NOS, Next Of Stack).

Adresní sběrnice (a tedy i programový čítač) má 13 bitů (může tedy adresovat paměť o velikosti 8192 šestnáctibitových slov). Programátor může pracovat s jednotlivými bajty a přistupovat k paměti jako k 16 kB prostoru, ale interně probíhá přístup vždy zarovnaný na 16 bitů.

Sada instrukcí obsahuje pět základních typů instrukcí pro zásobníkové procesory, totiž *literály* (konstanty, které se mají uložit na zásobník), skoky, podmíněné skoky, volání podprogramu a aritmetické / logické operace.

Literály jsou patnáctibitové, šestnáctý bit je 0, takže rozsah hodnot je 0 – 32767. Pokud je potřeba zadat číslo z rozsahu 32768 – 65535, musí překladač vložit instrukci „invert", která nastaví nejvyšší bit na 1.

Všechny cílové adresy skoků jsou 13bitové a absolutní. Podmíněný skok má pouze jeden jediný typ podmínky, totiž „skok, pokud je nejvyšší hodnota na zásobníku nula".

Šestnáctibitové instrukční slovo obsahuje několik pevně daných polí s jednoznačným významem:

- Bit 15: 1 = literál (pak ostatních 15 bitů obsahuje konstantu), 0 = instrukce
- Bity 14 a 13: pokud je bit 15=0, kódují typ instrukce:
    - 00 – nepodmíněný skok JUMP
    - 01 – podmíněný skok JUMPZ
    - 10 – volání podprogramu CALL
    - 11 – operace s ALU

U operací skoků a volání je v bitech 0 – 12 požadovaná adresa.

Instrukce ALU má v bitech 0 – 11 zakódováno, co má procesor udělat s daty (bit 12 není použitý – *v dokumentaci je chybně uvedeno, že je to bit 4*):

| Bity | Označení | Význam |
|---|---|---|
| **12** | - | Není použito |
| **11 – 8** | T' | Konkrétní operace s ALU |
| **7** | R->PC | Zkopírovat nejvyšší hodnotu ze zásobníku návratových adres (R) do čítače PC |
| **6-4** | func | Požadovaná operace |
| **3-2** | rstack | O kolik položek se má změnit zásobník R (se znaménkem) |
| **1-0** | dstack | O kolik položek se má změnit uživatelský datový zásobník (se znaménkem) |

Tato struktura instrukčního slova umožňuje provést některé optimalizace a spojit často používané operace dohromady. Například bit 7 (R->PC) slouží jako příznak návratu z podprogramu (vezme hodnotu ze zásobníku adres a vloží ji do PC). Není potřeba speciální instrukce, stačí u poslední instrukce podprogramu nastavit bit 7 a bity rstack na „-1" a procesor se správně vrátí za příslušnou instrukci call.

Bity 6-4 kódují požadovanou operaci, kterou má procesor vyvolat, a to takto:

| Hodnota | funkce |
|---|---|
| **000** | Nic |
| **001** | T->N |
| **010** | T->R |
| **011** | MWR (Memory Write) |
| **100** | IOWR (IO Write) |

ALU nabízí 16 operací, které jsou komplexnější než ty, co jsme používali u procesoru MHRD, ovšem také samotná aritmeticko-logická jednotka není složená z připravených obvodů, ale zapsaná behaviorálně, jak uvidíme dál.

| Kód | Výsledek | Kód | Výsledek |
|---|---|---|---|
| **0** | T | 8 | N < T (porovnání se znaménkem) |
| **1** | N | 9 | N >> T (posun doprava) |
| **2** | T+N | 10 | N << T (posun doleva) |
| **3** | T AND N | 11 | R (hodnota z RS) |
| **4** | T OR N | 12 | [T] (čtení z paměti) |
| **5** | T XOR N | 13 | [T(io)] (čtení z periferie) |
| **6** | NOT T | 14 | Počty položek na RS a DS |
| **7** | N = T (porovnání) | 15 | N u< T (porovnání bez znaménka) |

Pro nejzákladnější operaci se zásobníkem, DUP, tedy nastavíme slovo takto: Výsledek ALU bude hodnota T, dstack se zvýší o 1 a provádí se operace T->N.

## 16.2 Implementace procesoru J1 ve Verilogu

Začneme společnými definicemi. Vytvoříme soubor common.h, který budeme později vkládat, kde bude zapotřebí.

```
`default_nettype none
`define WIDTH 16  -- Autor upravil i pro 32 bitů
`define DEPTH 4
```

Procesor obsahuje dva zásobníky, a protože jsou to reálné paměťové struktury, pojďme si je nadeklarovat jako moduly:

```
`include "common.h"

module stack
  #(parameter DEPTH=4)
  (input wire clk,
  /* verilator lint_off UNUSED */
  input wire resetq,
  /* verilator lint_on UNUSED */
  input wire [DEPTH-1:0] ra,
```

```
  output wire [`WIDTH-1:0] rd,
  input wire we,
  input wire [DEPTH-1:0] wa,
  input wire [`WIDTH-1:0] wd);

  reg [`WIDTH-1:0] store[0:(2**DEPTH)-1];

  always @(posedge clk)
    if (we)
      store[wa] <= wd;

  assign rd = store[ra];
endmodule
```

Modul je parametrizovaný jedním parametrem, DEPTH (defaultní hodnota je 4). Parametr udává počet bitů adresy, tedy položek na zásobníku (těch je $2^{DEPTH}$). Přednastavená hodnota 4 tedy znamená, že zásobník je pro 16 položek. Zásobník má dvě adresní sběrnice a dvě datové sběrnice – jeden pár pro čtení, jeden pro zápis. Vstup we (Write Enable) povoluje zápis – pokud je aktivní, zásobník zapíše hodnotu z datového vstupu wd na adresu ze vstupu wa. Na výstupu rd je stále obsah paměti z adresy ra.

Zásobník sám o sobě neobsahuje žádnou další logiku, která by se starala o ukazatel, o jeho zvyšování a snižování a o podobné činnosti.

A takhle vypadá celá implementace procesoru, jak ji publikoval autor James Bowman: https://github.com/jamesbowman/j1

Dovolím si ji okomentovat a zdůraznit zajímavé pasáže:

```
`include "common.h"

module j1(
   input wire clk,
   input wire resetq,

   output wire io_wr,
   output wire [15:0] mem_addr,
   output wire mem_wr,
   output wire [`WIDTH-1:0] dout,
   input  wire [`WIDTH-1:0] mem_din,
```

```
input  wire [`WIDTH-1:0] io_din,

output wire [12:0] code_addr,
input  wire [15:0] insn
);
```

Port procesoru kromě „povinné jízdy“, tedy hodinového vstupu a RESETu, obsahuje sběrnice pro čtení instrukcí (code_addr posílá adresu, insn je šestnáctibitová instrukce, přečtená z paměti) a pro práci s pamětí dat a periferií.

Paměť dat je opravdu pouze pro data, zásobníky jsou implementovány hardwarově uvnitř.

Procesor posílá 16bitovou adresu paměti mem_addr. Zapisovaná data posílá po sběrnici dout. Pomocí signálů mem_wr a io_wr říká, zda se zapisuje do paměti, nebo do prostoru periferií. Paměti a periferie naopak přivádějí data po sběrnicích mem_din a io_din.

Následují deklarace ukazatelů a přístupových signálů pro zásobníky. Datový zásobník má registr *dsp* (Data Stack Pointer) a registr *st0* (hodnota na vrcholu zásobníku). Tyto registry jsou zdvojené – pracovní registry udržují hodnoty, ke kterým se v aktuálním cyklu přistupuje. Pokud instrukce vyvolá změnu, mění se hodnota registru dspN (Data Stack Pointer New) nebo st0N, a ta je na konci cyklu zkopírována do pracovních registrů.

```
reg [`DEPTH-1:0] dsp;     // Data stack pointer
reg [`DEPTH-1:0] dspN;
reg [`WIDTH-1:0] st0;     // Top of data stack
reg [`WIDTH-1:0] st0N;
reg dstkW;                 // D stack write


reg [`DEPTH-1:0] rsp, rspN;
reg rstkW;                // R stack write
wire [`WIDTH-1:0] rstkD;  // R stack write value
```

Signály *dstkW* a *rstkW* slouží k zapsání hodnoty na zásobníky. U zásobníku R se zapisovaná hodnota přivádí po datové sběrnici *rstkD*.

```
reg [12:0] pc, pcN;
```

Program counter a jeho pracovní zrcadlo.

```
reg reboot = 1;
wire [12:0] pc_plus_1 = pc + 1;
```

Signál *pc_plus_1* obsahuje přesně tu hodnotu, kterou čekáte, a slouží jako pomocný signál pro nejčastější případ, totiž že je nutno zvýšit čítač adres instrukcí.

```
assign mem_addr = st0N[15:0];
assign code_addr = {pcN};
```

Jako adresa do paměti dat je použita hodnota TOS na datovém zásobníku (resp. dolních 16 bitů), adresa do paměti kódu je připojena na registr pcN.

```
wire [`WIDTH-1:0] st1, rst0;
```

Signál *st1* vlastně odpovídá FORTHovskému NOS (Next On Stack), tedy položce pod aktuální položkou. Aktuální položka je TOS a je v registru *st0*.

```
stack #(.DEPTH(`DEPTH))
 dstack(.clk(clk), .resetq(resetq), .ra(dsp), .rd(st1), .we(dstkW), .wa(dspN), .wd(st0));
```

Instancujeme modul stack s hloubkou danou parametrem DEPTH. Hodiny a RESET jsou samozřejmost (RESET tedy nemá žádný efekt). Čtecí adresa je daná ukazatelem dsp, zapisovací adresa je dspN (logicky: zapisuje se na konci cyklu na adresu, kterou si v rámci instrukce teprve spočítáme). Přečtená data tvoří signál st1, data k zápisu jsou daná registrem st0. Zápis povoluje signál dstkW.

```
stack #(.DEPTH(`DEPTH))rstack(.clk(clk), .resetq(resetq), .ra(rsp), .rd(rst0), .we(rstkW),
.wa(rspN), .wd(rstkD));
```

Až na konkrétní signály je definice zásobníku R téměř totožná.

Teď začíná ta nejvíc *cool* pasáž, totiž proces, který počítá novou hodnotu na zásobníku, a to na základě načteného instrukčního kódu. Všimněte si, že rozhoduje pouze horních 8 bitů (15 až 8). Pokud má tvar „1xxx_xxxx“, jde o zápis literálu, tj. konstanty, která se má přenést do st0. Takže je doplněna nulami zleva na plnou šířku datového slova.

U instrukcí „000x“ a „010x“ (JUMP a CALL) se stav st0 nemění, u podmíněného skoku se načte hodnota NOS (podmíněný skok „zkonzumuje“ hodnotu TOS).

U instrukcí s ALU („011x_iiii“) se podle příslušných bitů „i“ zvolí požadovaná operace.

```
  always @*
  begin
    // Compute the new value of st0
    casez ({insn[15:8]})
      8'b1??_?????: st0N = { {(`WIDTH - 15){1'b0}}, insn[14:0] };    // literal
      8'b000_?????: st0N = st0;  // jump
      8'b010_?????: st0N = st0;  // call
      8'b001_?????: st0N = st1;  // conditional jump
      8'b011_?0000: st0N = st0;  // ALU operations...
      8'b011_?0001: st0N = st1;
      8'b011_?0010: st0N = st0 + st1;
      8'b011_?0011: st0N = st0 & st1;
      8'b011_?0100: st0N = st0 | st1;
      8'b011_?0101: st0N = st0 ^ st1;
      8'b011_?0110: st0N = ~st0;
      8'b011_?0111: st0N = {`WIDTH{(st1 == st0)}};
      8'b011_?1000: st0N = {`WIDTH{($signed(st1) < $signed(st0))}};
`ifdef NOSHIFTER
// `define NOSHIFTER in common.h to cut slice
// usage in half and shift by 1 only
      8'b011_?1001: st0N = st1 >> 1;
      8'b011_?1010: st0N = st1 << 1;
`else
// otherwise shift by 1-any number of bits
      8'b011_?1001: st0N = st1 >> st0[3:0];
      8'b011_?1010: st0N = st1 << st0[3:0];
`endif
      8'b011_?1011: st0N = rst0;
      8'b011_?1100: st0N = mem_din;
      8'b011_?1101: st0N = io_din;
      8'b011_?1110: st0N = {{(`WIDTH - 8){1'b0}}, rsp, dsp};
      8'b011_?1111: st0N = {`WIDTH{(st1 < st0)}};
      default: st0N = {`WIDTH{1'bx}};
    endcase
  end
```

Všimněte si znaku „?“ jako *placeholderu* pro hodnotu, která nás v tu chvíli nezajímá.

*Pokud v common.h definujeme konstantu `define NOSHIFTER, můžeme ušetřit logické buňky a místo shiftování až o 16 pozic (uvažují se 4 bity ST0) nadefinovat operaci posunu tak, že posouvá vždy o jediný bit.*

```
wire func_T_N =   (insn[6:4] == 1);
wire func_T_R =   (insn[6:4] == 2);
wire func_write = (insn[6:4] == 3);
wire func_iow =   (insn[6:4] == 4);
```

Bity 6, 5 a 4 instrukčního slova kódují požadovanou operaci, jak jsme si popisovali dříve.

```
wire is_alu = (insn[15:13] == 3'b011);
```

Signál is_alu je 1, pokud instrukce byla typu „011“, tedy operace s daty (ne skoky ani literál).

```
assign mem_wr = !reboot & is_alu & func_write;
assign dout = st1;
```

Signál pro zápis do externí paměti přijde ve chvíli, kdy není stav reboot, proběhla operace s ALU a zároveň byla požadována funkce „write“. Data, která se zapisují, jsou vždy v NOS (tedy druhá nejvyšší položka na zásobníku).

```
assign io_wr = !reboot & is_alu & func_iow;
```

Pro zápis do periferií platí totéž.

```
assign rstkD = (insn[13] == 1'b0) ? {{(`WIDTH - 14){1'b0}}, pc_plus_1, 1'b0} : st0;
```

Data pro zápis do RS se liší podle stavu bitu 13 instrukčního slova. Pokud je 0 (mají ji instrukce JUMP a CALL), připraví se hodnota PC+1, pokud je 1 (mají ji JUMPZ nebo ALU), připraví se TOS.

*Může se stát, že instrukce obsahuje literál, který má bit 13 nastavený nebo vynulovaný – je to jedno, protože tyto instrukce nevyvolají zápis do RS. Stejně tak instrukce JMP – sice připraví návratovou adresu, ale nezapíše ji.*

```
reg [`DEPTH-1:0] dspI, rspI;
```

Hodnoty *dspI* a *rspI* představují *inkrement* ukazatele DS, resp. RS. Nejčastěji bývá jedničkový (do zásobníku přibyla hodnota), nulový (zásobník nemění svůj stav), nebo minus jedna (ze zá-

sobníku ubyla hodnota).

```
always @*
begin
  casez ({insn[15:13]})
  3'b1??:   {dstkW, dspI} = {1'b1,      4'b0001};
  3'b001:   {dstkW, dspI} = {1'b0,      4'b1111};
  3'b011:   {dstkW, dspI} = {func_T_N,  {insn[1], insn[1], insn[1:0]}};
  default:  {dstkW, dspI} = {1'b0,      4'b0000};
  endcase
  dspN = dsp + dspI;
```

Podle typu instrukce se nastavuje inkrement pro ukazatel DS a zároveň signál zápisu do DS. Instrukce „literál“ (1xx) nastaví zápis na 1 a inkrement jedničkový. Podmíněný skok konzumuje hodnotu z datového zásobníku, takže zápisový bit je 0 a inkrement -1. Pro ALU se zápis řídí příznakem požadavku T->N a inkrement je dán bity 1 a 0 instrukčního slova (*dstack*). V ostatních případech se hodnota nemění a nic se nezapisuje.

```
  casez ({insn[15:13]})
  3'b010:   {rstkW, rspI} = {1'b1,      4'b0001};
  3'b011:   {rstkW, rspI} = {func_T_R,  {insn[3], insn[3], insn[3:2]}};
  default:  {rstkW, rspI} = {1'b0,      4'b0000};
  endcase
  rspN = rsp + rspI;
```

Podobná logika řídí i zásobník návratových adres RS. Instrukce CALL (010) zvyšuje ukazatel RS o 1 a zapisuje (rstkW=1). Instrukce ALU zapisuje, pokud aktivuje funkci T->R, a změna ukazatele se řídí bity 3 a 2 instrukčního slova (*rstack*).

```
  casez ({reboot, insn[15:13], insn[7], |st0})
  6'b1_???_?_?:   pcN = 0;
  6'b0_000_?_?,
  6'b0_010_?_?,
  6'b0_001_?_0:   pcN = insn[12:0];
  6'b0_011_1_?:   pcN = rst0[13:1];
  default:        pcN = pc_plus_1;
  endcase
end
```

Nepřekvapí vás, že se podobně jednoduše řeší i hodnota PC. Její nová hodnota je většinou PC+1, s několika výjimkami. Pokud je „reboot = 1“, je nová hodnota PC rovna nule. Pro instrukce

CALL a JUMP je nová hodnota PC rovna bitům 13 – 0 z instrukčního slova. Pro instrukci JUMPZ to platí, pokud je st0 nulový. Pokud jde o instrukci ALU a má nastavený bit 7 (RET), nastaví se hodnota ze zásobníku návratových adres.

```
  always @(negedge resetq or posedge clk)
  begin
    if (!resetq) begin
      reboot <= 1'b1;
      { pc, dsp, st0, rsp } <= 0;
    end else begin
      reboot <= 0;
      { pc, dsp, st0, rsp } <= { pcN, dspN, st0N, rspN };
    end
  end
endmodule
```

A toto je poslední proces procesoru J1. Je aktivní při sestupné hraně signálu resetq nebo při náběžné hraně hodin. Při sestupné hraně se nastavuje reboot na 1 a nulují se registry pc, dsp, rsp a st0.

Při náběžné hraně hodin se nuluje reboot a hodnoty pc, dsp, rsp a st0 se nastavují podle svých zrcadlových registrů pcN, dspN, rspN a st0N na novou hodnotu.

*Příklady použití, stejně jako definice základních FORTHových slov, najdete na autorově stránce: https://www.excamera.com/sphinx/fpga-j1.html*

## 16.3 Verilog vs VHDL

Tato kapitola tady nesmí chybět. Stejně jako u jiných věčných sporů vývojářského světa i zde platí, že čím je spor zbytečnější a malichernější, o to vášnivější zastánce má. Já jsem se přiklonil celou knihou trochu na stranu VHDL, ale je fér zmínit reálné rozdíly a nechat vás, ať si uděláte obrázek a zvolíte si stranu, kterou budete zastávat, až se vás svět zeptá: „*Losnu, nebo Mažňáka?*“

Nejprve se podívejme na koncepční rozdíly. Verilog i VHDL pokrývají velkou část spektra vývoje systémů, od jednotlivých hradel přes logické celky, RTL a behaviorální popis až k popisu celého systému. Verilog jde o kousek níž a chybí mu vyšší celky (tam zasahuje SystemVerilog), VHDL jde o kousek výš, a naopak mu chybí možnosti modelace hradel jako takových (na to existuje nástroj VITAL).

VHDL navádí k tomu udržovat jednotlivé komponenty v samostatných souborech (nemusíte, ale je to lepší), které lze kompilovat odděleně. Verilog naproti tomu nezapře své „programátorské“ kořeny, a tak v něm velmi záleží na pořadí kompilování modulů.

VHDL je silně typovaný jazyk, takže pro převod mezi dvěma „skoro stejnými“ typy potřebujete často explicitní konverzní funkci. Navíc umožňuje uživateli definovat vlastní typy. Na druhou stranu díky tomu už ve fázi kompilace odhalíte některé problémy či chyby. Verilog je v tomto výrazně jednodušší, typy definuje sám, což může pro někoho představovat výhodu.

VHDL umožňuje vytváření balíčků procedur a funkcí, které jsou znovupoužitelné. Verilog naproti tomu používá koncepci „includování“ souborů.

Verilog je výrazně snazší se naučit, hlavně pokud máte základ z programovacích jazyků, odvozených od C. Dokonce existuje i nástroj (zvaný PLI), který umožňuje psát části kódu v nativním C/C++. VHDL vyžaduje mnohem víc teorie a nezbytných pravidel v úvodu (*křivka učení* roste zpočátku velmi pomalu).

VHDL nabízí generické konstrukce, Verilog je nemá (kromě parametrizace modulů).

VHDL má zabudované pouze základní logické funkce NOT, AND, OR, NAND, NOR, XOR, XNOR, a to pro jeden či dva vstupy. Pokud chcete modelovat např. zpoždění signálu, musíte používat klauzule, pokud chcete vlastní hradla, použijte VITAL. Verilog naproti tomu umožňuje modelování jednotlivých hradel či buněk FPGA.

VHDL je mnohem víc „upovídaný jazyk“ – tam, kde si Verilog vystačí se speciálními znaky nebo nezbytně nutnými symboly, tam se ve VHDL můžete téměř upsat. Kód ve VHDL je proto rozsáhlejší a méně kompaktní než ve Verilogu.

| | **VHDL** | **Verilog** |
|---|---|---|
| **Koncepce** | ADA, nerozlišuje velká / malá písmena | C |
| **Typování** | Silné | Slabé (implicitní konverze typů) |
| **Procesy citlivé na „vše“** | process(all) – od verze 2008 | @(*) |
| **Logické hodnoty** | 9 | 4 |
| **Uživatelské typy** | Ano | Ne |
| **Zaměření** | Bezpečný vývoj | Rychlý vývoj |

# 17 Doslov

## 17 Doslov

Jsou to tři roky (a pár měsíců) od doby, kdy jsem sebral odvahu a rozhodl se, že napíšu knížku o elektronice pro úplné začátečníky. Ne pro děti, ale pro nadšence, co třeba objevili Arduino a cítili, že jim chybí nějaká část teorie do skládačky. Třeba proč něco nesvítí, když by mělo, nebo proč je to rozžhavené, když o tom v manuálu nic nepsali...

S myšlenkou jsem si pohrával už delší dobu. Chtěl jsem původně začít od voltů a elektronů, projít přes tranzistory, elementární logické obvody, složitější konstrukce, pak se zastavit u počítačů a starých procesorů, ukázat, jak se takové věci konstruovaly, a skončit téměř u současného high-endu, totiž u programovatelných polí. Jenže téma se ukázalo tak obšírné, že z toho jsou nakonec knihy tři.

Mým cílem nebylo psát učebnice, od toho jsou tu jiní, povolanější a exaktnější autoři. Chtěl jsem spíš přiblížit svět číslicové techniky nadšeným lidem, kteří možná nemají teoretické znalosti a nezbytný základ, ale mají chuť a touhu učit se zajímavé věci (a ty poslední dvě slova podtrhnu).

Proto jsem kladl větší důraz na to, abych v popisu nepostupoval cestou „od teoretických základů přes teoretické podrobnosti k prvnímu praktickému pokusu", jak je běžnější, ale „od experimentu k jeho vysvětlení a pochopení", který osobně považuju za vhodnější pro lidi, co se učit nemusí, ale chtějí. Doufám, že se mi to alespoň zčásti podařilo, a děkuju všem čtenářům, kteří tento přístup ocenili.

Touto knihou se tedy uzavřela „elektronická trilogie", věnovaná číslicové technice. Ne že by už nebylo o čem psát – například jsem opominul, záměrně, obrovskou oblast vysokofrekvenčních obvodů, z analogové techniky jsem zase nakousl jen nezbytné minimum. Ale při psaní jsem měl stále před očima vás, čtenáře. Vidím vás, a jistě se nebudete zlobit, jako lidi, kteří hledají něco zajímavého, co by si sami zkusili udělat. Láká vás elektronika, třeba jste si zkusili něco s Arduinem, a říkáte si: *Kam dál? Co víc? Kam se posunout? Co ještě zajímavého zkusit?* Proto jsem chtěl zůstat u jednoho tématu a nezahlcovat text přemírou souvisejících informací, i s rizikem, že budu muset použít „black boxy" a některé věci předkládat stylem „tak to je, nebudeme to řešit, na vysvětlení není prostor..." Radši budu v textu nepřesný a neúplný, než nudný a suchopárný!

Možná se ještě u nějakého titulu setkáme. Někteří čtenáři zmiňovali, že by je zajímala radiotechnika, nostalgičtí fanoušci zmiňovali, že by chtěli něco jako *konstrukce* s *tranzistory*, co si pamatují z knih svého mládí, ale s moderními součástkami... Možností a námětů do budoucna je určitě dost.

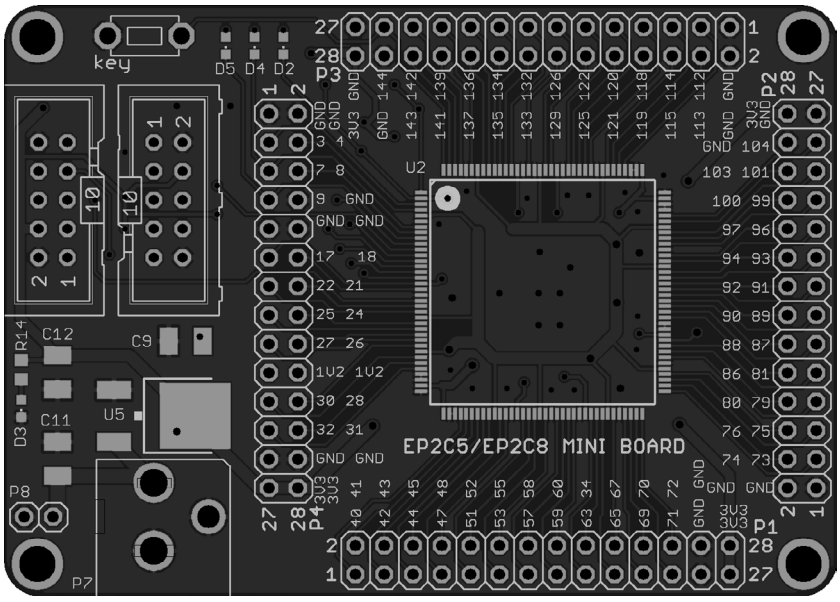
Kniha tedy končí – ale vy teprve začínáte. Ať se vám na vaší cestě za radostí z poznání daří!

Martin Malý

v Praze, 22. června 2020

# 18 Příloha: Kit EP2C5T144

# 18 Příloha: Kit EP2C5T144




## 18.1 Mapa obsazených pinů

| Pin | Periferie |
|---|---|
| **3** | LED D2 |
| **7** | LED D4 |
| **9** | LED D5 |
| **17** | Hodiny 50 MHz |
| **73** | Pull-up rezistor 10 K (k VCC) a kondenzátor 10M (k zemi) |
| **144** | Tlačítko KEY, spínané k zemi |

# 19 Příloha: Kit OMDAZZ

# 19 Příloha: Kit OMDAZZ

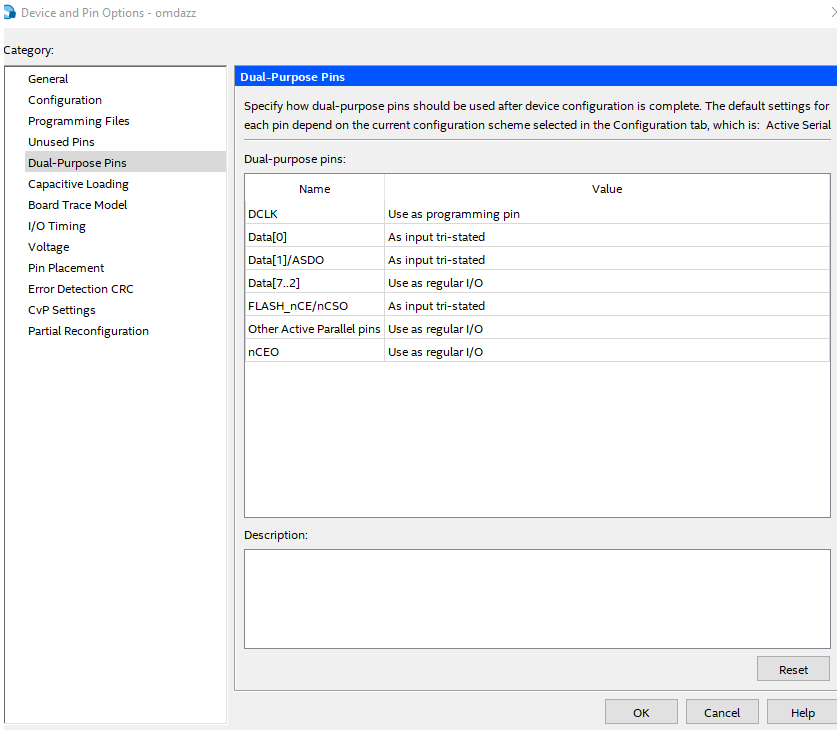
K tomuto kitu získáte od výrobce schéma (a pokud budete mít štěstí, bude v něm i anglický popis součástí) a přiřazení pinů. Já jsem si jej přepsal do podoby textového souboru – naleznete jej opět ve zdrojových kódech ke knize.

Srdcem kitu je obvod EP4CE6E22C8N, tedy nejmenší zástupce rodiny Cyclone IV. K němu je připojena konfigurační paměť, nikoli originál, ale sériová FLASH Winbond. Další „velký" obvod je čip dynamické paměti SDRAM s kapacitou 4M x 16 bitů (bližší informace najdete v kapitole o SDRAM).

Na kitu jsou čtyři LED, čtyři tlačítka (paralelně spojená se čtyřpólovým DIP přepínačem), bzučák, sériová paměť AT24C08 a teplotní čidlo LM75 (obojí na sběrnici $I^2C$), čtyřmístný sedmisegmentový displej LED, konektory pro RS-232, VGA, PS/2 a LCD displeje 1602 / 12864, a dokonce i infračervený přijímač (to kdybyste si chtěli zkusit ovládat něco dálkovým ovladačem od TV).

Okolo FPGA jsou tři pinové lišty (2x10, 2x19 a 2x22 pinů), k nimž je připojena většina dostupných pinů. Bohužel, špatná zpráva je, že všechny piny jsou obsazené, takže když budete chtít připojovat vlastní zařízení, nemáte moc možností. Můžete využít nanejvýš konektor pro LCD, nebo sběrnici $I^2C$, popřípadě piny pro VGA.

*Pozor! Při přiřazování signálu VGA_HSYNC pinu 101 vám Quartus oznámí chybu, totiž přiřazení druhého signálu již přiřazenému pinu. Problém je v tom, že pin 101 má funkci nCEO, která se používá při programování. Řešení je snadné – jděte do menu Assignments – Device, klikněte na tlačítko „Device and Pin Options", a v záložce „Dual-Purpose Pins" vyberte nCEO a zvolte „Use as regular I/O".*


Device and Pin Options - omdazz
Category:
General
Configuration
Programming Files
Unused Pins
Dual-Purpose Pins
Capacitive Loading
Board Trace Model
I/O Timing
Voltage
Pin Placement
Error Detection CRC
CvP Settings
Partial Reconfiguration
Dual-Purpose Pins
Specify how dual-purpose pins should be used after device configuration is complete. The default settings for each pin depend on the current configuration scheme selected in the Configuration tab, which is: Active Serial
Dual-purpose pins:
Name
Value
DCLK
Use as programming pin
Data[0]
As input tri-stated
Data[1]/ASDO
As input tri-stated
Data[7..2]
Use as regular I/O
FLASH_nCE/nCSO
As input tri-stated
Other Active Parallel pins
Use as regular I/O
nCEO
Use as regular I/O
Description:
Reset
OK
Cancel
Help

# 20 Příloha: VHDL v kostce

# 20 Příloha: VHDL v kostce

## 20.1 Operátory

Seřazeno od nejvyšší priority. Pokud není uveden typ výsledku, je jeho typ shodný s operandem, popř. u binárních operátorů s levým operandem.

| Operátor | Význam | Operand(y) | Typ výsledku |
|---|---|---|---|
| ** | mocnina | numeric ** integer | numeric |
| abs | absolutní hodnota | numeric | |
| not | negace | logic, boolean | |
| * | násobení | numeric | |
| / | dělení | numeric | |
| mod | zbytek po dělení (modulo) [1)] | integer mod integer | |
| rem | zbytek po dělení [1)] | integer rem integer | |
| + | unární plus | + numeric | |
| - | unární minus | - numeric | |
| + | sčítání | numeric + numeric | |
| - | odčítání | numeric – numeric | |
| & | spojování | pole nebo element & pole nebo element | pole |
| sll, srl<br>sla, sra,<br>rol, ror | posun vlevo / vpravo logický, aritmetický a rotace | SLV rol integer | |
| =, /=,<br><, >,<br><=, >= | relační operátory (porovnání) | | boolean |

| | | | |
|---|---|---|---|
| and, or,<br>nand, nor,<br>xor, xnor | binární logické funkce | SLV nebo boolean | |

Pozn. 1: rozdíl mezi mod a rem je v tom, že výsledek mod má znaménko dělitele, rem znaménko dělence. Příklad:

| **Dělenec** | **Dělitel** | **mod** | **rem** |
|---|---|---|---|
| **9** | 5 | 4 | 4 |
| **9** | -5 | -1 | 4 |
| **-9** | 5 | 1 | -4 |
| **-9** | -5 | -4 | -4 |

## 20.2 Atributy

T je libovolný typ, A je pole, S signál a E entita

| **Atribut** | **Význam** |
|---|---|
| **T'BASE** | základní typ typu T |
| **T'LEFT** | levá hodnota T (vyšší u downto) |
| **T'RIGHT** | pravá hodnota T (nižší u downto) |
| **T'HIGH** | nejvyšší hodnota T |
| **T'LOW** | nejnižší hodnota T |
| **T'ASCENDING** | true, pokud je typ T definovaný jako vzestupný (to) |
| **T'IMAGE(X)** | konverze hodnoty X typu T na řetězec |
| **T'VALUE(X)** | konverze řetězce X na hodnotu typu T |
| **T'POS(X)** | pozice prvku X v typu T (od 0) |

| | |
|---|---|
| **T'VAL(X)** | X-tý prvek v typu T |
| **T'SUCC(X)** | prvek následující po prvku X v typu T |
| **T'PRED(X)** | prvek předcházející prvku X v typu T |
| **T'LEFTOF(X)** | prvek nalevo od prvku X v typu T |
| **T'RIGHTOF(X)** | prvek napravo od prvku X v typu T |
| **A'LEFT** | levý krajní index pole A |
| **A'LEFT(N)** | levý krajní index pole A v dimenzi N |
| **A'RIGHT** | pravý krajní index pole A |
| **A'RIGHT(N)** | pravý krajní index pole A v dimenzi N |
| **A'HIGH** | vyšší index pole A |
| **A'HIGH(N)** | vyšší index pole A v dimenzi N |
| **A'LOW** | nižší index pole A |
| **A'LOW(N)** | nižší index pole A v dimenzi N |
| **A'RANGE** | rozsah pole A (např. „7 downto 0“) |
| **A'RANGE(N)** | rozsah pole A v dimenzi N |
| **A'REVERSE_RANGE** | obrácený rozsah pole A |
| **A'REVERSE_RANGE(N)** | obrácený rozsah pole A v dimenzi N |
| **A'LENGTH** | počet prvků pole A |
| **A'LENGTH(N)** | počet prvků pole A v dimenzi N |
| **A'ASCENDING** | true, pokud je rozsah pole A definovaný jako vzestupný (to) |
| **A'ASCENDING(N)** | true, pokud je rozsah pole A v dimenzi N definovaný jako vzestupný (to) |
| **S'DELAYED(t)** | vytvoří signál, který je vůči zdroji opožděný o zadaný interval |
| **S'STABLE** | true, pokud je signál stabilní, tj. nemění se |

| | |
|---|---|
| **S'STABLE(t)** | true, pokud se signál nemění po zadaný časový úsek |
| **S'QUIET** | true, pokud nemá signál žádné změny (ani naplánované) |
| **S'QUIET(t)** | true, pokud signál nemá ani naplánované změny po zadaný časový úsek |
| **S'TRANSACTION** | hodnota typu bit, která se mění při každé změně signálu |
| **S'EVENT** | true, pokud se hodnota signálu v simulačním cyklu změnila |
| **S'ACTIVE** | true, pokud se v simulačním cyklu změnila hodnota, nebo byla změna naplánována |
| **S'LAST_EVENT** | čas od poslední události v signálu |
| **S'LAST_ACTIVE** | čas od poslední aktivity |
| **S'LAST_VALUE** | předchozí hodnota signálu |
| **E'SIMPLE_NAME** | jméno entity E jako řetězec |
| **E'INSTANCE_NAME** | kompletní jméno entity E včetně hierarchie |
| **E'PATH_NAME** | cesta k entitě E |

## 20.3 Deklarace

### Nekompletní deklarace typu

Deklaruje, že identifikátor označuje nějaký typ, aniž by bylo specifikováno, jaký. Přesná definice se musí objevit někde jinde.

**type** identifikátor;

```
type node;
```

### Deklarace skalárního typu

Deklaruje typ, který může být použit všude tam, kde je potřeba skalární (jednorozměrný) údaj.

**type** identifikátor **is** definice_skalárního_typu;

```
type my_small is range -5 to 5;
type my_bits  is range 31 downto 0;
type my_float is range 1.0 to 1.0E6;
```

### Deklarace složeného typu

Deklaruje typ pole, záznam nebo jednotka (units).

```
type word is array (0 to 31) of bit;
type data is array (7 downto 0) of word;
type mem is array (natural range <>) of word;
type matrix is array (integer range <>,
                      integer range <>) of real;


type zaznam is
   record
       I : integer;
       X : real;
       day : integer range 1 to 31;
       name : string(1 to 48);
       prob : matrix(1 to 3, 1 to 3);
   end record;

type uzel is -- binarni strom
   record
       key   : string(1 to 3);
       data  : integer;
       left  : uzel_ukaz;
       right : uzel_ukaz;
       color : color_type;
   end record;


 type delka is range 0 to 1E16
    units
      Ang;                 -- angstrom
      nm   = 10 Ang;       -- nanometr
      um   = 1000 nm;      -- mikrometr (mikron)
      mm   = 1000 um;      -- milimetr
      cm   = 10 mm;        -- centimetr
```

```
    dm   = 100 mm;        -- dekametr
    m    = 1000 mm;       -- metr
    km   = 1000 m;        -- kilometr

    mil  = 254000 Ang;    -- mil (1/1000 palce)
    inch = 1000 mil;      -- palec
    ft   = 12 inch;       -- stopa
    yd   = 3 ft;          -- yard
    fthn = 6 ft;          -- fathom
    frlg = 660 ft;        -- furlong
    mi   = 5280 ft;       -- míle
    lg   = 3 mi;          -- league
end units;
```

### Deklarace ukazatele

Deklaruje typ, použitelný k odkazování na jiné typy.

**type** identifikátor **is access** identifikátor_typu;

```
type uzel_ukaz is access uzel;
variable koren : uzel_ukaz := new uzel'("xyz", 0, null, null, red);
variable polozka : node := koren.all;
```

### Deklarace typu soubor

Deklaruje typ pro přístup k datovým souborům.

```
type my_text is file of string ;

type word_file is file of word ;

file output : my_text;
file_open(output, "my.txt", write_mode);
write(output, "some text"&lf);
file_close(output);

file test_data : word_file;
file_open(test_data, "test1.dat", read_mode);
read(test_data, word_value);
```

### Deklarace podtypu

Deklaruje typ, který je podtypem (subtypem) nějakého komplexnějšího typu.

```
subtype name_type is string(1 to 20) ;
variable a_name : name_type := "Martin Maly         ";

subtype small_int is integer range 0 to 10 ;
variable little : small_int := 4;

subtype word is std_logic_vector(31 downto 0) ;
signal my_word : word := x"FFFFFFFC";
```

### Deklarace konstant

Konstanta je pojmenovaná hodnota, kterou nelze změnit. Jakmile ji jednou nadeklarujete, je její hodnota všude stejná.

**constant** identifikátor : typ := výraz;

```
constant Pi : real := 3.14159;
constant Half_Pi : real := Pi/2.0;
constant cycle_time : time := 2 ns;
constant N, N5 : integer := 5;
```

I deklarace konstanty může být „odložená“, bez části přiřazení hodnoty, např. „constant N: integer;“, ale taková deklarace může být použita pouze v deklaraci balíčku; v těle balíčku musí být její hodnota nastavena.

### Deklarace signálu

Deklaruje signál a může mu přiřadit výchozí hodnotu.

**signal** identifikátor : typ [druh_signálu] [ := hodnota ];

Druh signálu může být „register“ nebo „bus“, čímž explicitně říkáte, jak má syntetizér s takovým signálem nakládat, a pokud s ním budete pracovat v rozporu s jeho druhem, upozorní vás na to.

```
signal a_bit : bit := '0';
a_bit <= b_bit xor '1';
```

```
signal my_word : word := X"01234567";
my_word <= X"FFFFFFFF";

signal foo : word register; -- guarded signal
signal bar : word bus;      -- guarded signal
signal join : word wired_or; -- wired_or musí být rozhodovací funkce
```

Signál u nerozhodovaných (unresolved) typů může mít pouze jedno přiřazení, které definuje jeho hodnotu. „Bit“ je nerozhodovaný typ stejně jako „std_ulogic“, ale například „std_logic“ je typ rozhodovaný (resolved), takže mu může být v jednu chvíli přiřazeno víc budicích hodnot. Rozhodovací funkce (resolution function) pak musí nadefinovat, jaká hodnota bude použita.

### Deklarace proměnné

Platí totéž jako u deklarace proměnné.

**variable** identifikátor : typ [ := výraz ];

```
variable count : integer := 0;
count := count + 1;
```

Proměnné mohou být i sdíleny i mezi více procesy, ale v každém simulačním cyklu smí k proměnné přistupovat pouze jediný proces.

**shared variable** identifikátor : typ [ := výraz ];

```
shared variable status : status_type := stop;
status := start;
```

Proměnné, deklarované v podprogramech a procedurách, nesmí být deklarované jako sdílené. Naopak proměnné, deklarované v entitách, architekturách, balíčcích a blocích musí být sdílené.

*Pozor na syntaktickou odlišnost: proměnné i signály se inicializují v deklaraci pomocí :=, ale hodnoty se přiřazují jinak. Proměnným pomocí :=, signálům pomocí* <=

### Deklarace objektu typu soubor

S tímto druhem deklarace jsme se seznámili v knize. Deklaruje soubor nikoli jako typ, ale přímo jako objekt, ke kterému lze přistupovat.

**file** identifikátor : typ [ otevření_souboru];

Nepovinný parametr *otevření_souboru* má tvar:

[ **open** přístup ] **is** jméno_souboru;

Přístup může mít hodnotu read_mode, write_mode nebo append_mode (pro přidávání do souboru).

```
use STD.textio.all; -- pro práci se soubory
file my_file : text open write_mode is "soubor.dat";
variable my_line : line;

write(my_line, string'("Hello."));
writeline(my_file, my_line);
```

Jméno souboru musí respektovat konvenci, používanou operačním systémem, na kterém se simulace spouští. Můžete být kreativní, ale DOSový formát názvu 8.3 bez diakritiky je sázka na jistotu...

### Deklarace aliasu

Alias je jiné pojmenování pro už existující jméno typu, signálu nebo operátoru.

**alias** jméno **is** existující_jméno;

**alias** jméno [ : subtyp] **is** [ signatura ];

```
alias mantissa:std_logic_vector(23 downto 0) is my_real(8 to 31);
alias exponent is my_real(0 to 7);
alias "<" is my_compare [ my_type, my_type, return boolean ] ;
alias 'H' is STD.standard.bit.'1' [ return bit ] ;
```

### Deklarace atributu

Uživatelé mohou deklarovat vlastní atributy typům, signálům a dalším objektům VHDL.

**attribute** identifikátor : typ;

```
attribute enum_encoding : string; -- deklarace atributu
```

Později můžete specifikovat např. přiřazení hodnot výčtovému typu:

**attribute** identifikátor **of** jméno : třída **is** výraz;

Třída může být libovolná třída VHDL: **architecture, component, configuration, constant, entity, file, function, group, label, literal, package, procedure, signal, subtype, type, variable, units**.

```
type my_state is (start, stop, ready, off, warmup);
attribute enum_encoding of my_state : type is "001 010 011 100 111";
signal my_status : my_state := off; -- hodnota "100"
```

### Deklarace komponenty

Deklaruje rozhraní komponenty (obvodu).

**component** jméno **is**

[ **generic** ( generické proměnné ) ; ]

**port** ( vstupy_a_výstupy ) ;

**end component** [ jméno ] ;

Generické proměnné mají tvar *proměnná : typ := hodnota*;

Deklarace vstupů a výstupů mají tvar *jméno : směr typ*; kde *směr* je **in**, **out**, **inout**, **buffer** nebo **linkage**.

```
component delayLine is
     generic ( delay : time := 100 ps  );
     port ( input : in std_logic_vector(31 downto 0);
            output: out std_logic_vector(31 downto 0);
            load  : in  std_logic;
            clk   : in  std_logic );
end component delayLine;
```

Při použití se na komponentu odvoláváme jménem a mapováním proměnných (generic map, port map):

```
delay : delayLine
           generic map ( delay => 150 ps)
           port map ( input => dataIn,
```

```
                output => dataOut,
                    load => cs,
                    clk => sysClock);
```

Alternativní instanciace umožňuje vynechat deklaraci rozhraní entity a odkázat se pomocí zápisu

**entity Work.**jméno [ (architektura) ] [ **generic map...** ] **port map** ...

```
delay: entity WORK.delayLine(main)
            generic map (150 ps)
            port map ( dataIn, dataOut, cs, sysClock );
```

## 20.4 Rozhodování (resolution)

Rozhodovací funkce (resolution function) definuje u rozhodovaných (resolved) typů, jak z více zdrojů určit výslednou hodnotu signálu. Pokud k typu definujete rozhodovací funkci, můžete jej použít jako rozhodovaný typ a používat více zdrojů pro jeden signál. Rozhodovací funkce je pak vyvolána vždy, kdy je potřeba určit hodnotu.

Rozhodovací funkci můžete definovat i ke konkrétnímu signálu. Tato funkce je pak použita vždy, kdy je potřeba rozhodnout o hodnotě tohoto signálu, a signál má více zdrojů.

Rozhodovací funkce musí být čistá funkce s jediným vstupním parametrem, kterým je konstantní jednorozměrné pole hodnot, kterých mohou nabývat zdrojové signály.

V balíčku std_logic_1164 najdete příklad takové funkce, která rozhoduje hodnoty u typu std_logic. Nejprve je definovaný unresolved typ std_ulogic a jeho hodnoty:

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );

type std_ulogic_vector is array ( natural range <> ) of std_ulogic;
```

Následuje samotná rozhodovací funkce resolved, která vrací std_ulogic na základě buzení více signály std_ulogic. (Je to logické, protože k tomu, abychom mohli rozhodnout o výstupu, musíme mít rozhodnuté vstupy. Nejsou-li rozhodnuté, je potřeba nejprve rozhodnout je.

```
function resolved ( s : std_ulogic_vector ) return std_ulogic;
    variable result : std_ulogic := 'Z';  -- weakest state default
begin
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    if  s'length = 1 then
        return s(s'low);
    else
        for i in s'range loop
            result := resolution_table(result, s(i));
        end loop;
    end if;
    return result;
end resolved;
```

Všimněte si, že defaultní hodnota je „nejslabší možná“, tedy vysoká impedance (stav Z, tedy „signál odpojen“). Pokud je na vstupu jediný signál, je vrácena jeho nižší hodnota, pokud je víc budicích signálů, je jeden po druhém porovnán s pracovní hodnotou (defaultně Z) a výsledek srovnání těchto dvou hodnot je nová pracovní hodnota. K porovnání slouží dvourozměrné pole *resolution_table*:

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

V ní jsou vypsané hodnoty pro všechny možné kombinace vstupních hodnot.

S funkcí resolved je pak nadefinován podtyp std_logic jako „resolved std_ulogic“.

```
subtype std_logic is resolved std_ulogic;
```

## 20.5 Sekvenční příkazy

Sekvenční příkazy se používají v procesech, procedurách a funkcích. Jsou dostupné následující příkazy:

### wait

Pozastaví provádění příkazů.

[ návěstí: ] **wait** [ sensitivity ] [ podmínka ] ;

```
wait for 10 ns;              -- čekání po specifikovaný čas
wait until clk='1';          -- čekání na splnění podmínky
wait until A>B and S1 or S2; -- čekání na splnění podmínky
wait on sig1, sig2;          -- čekání na událost. Libovolná událost kteréhokoli signálu
ukončí čekání
```

### assert

Používá se ke kontrole hodnot při simulaci, popřípadě k vygenerování chybového hlášení.

[ návěstí: ] **assert** podmínka [ **report** string ] [ **severity** důležitost ] ;

```
assert a=(b or c);
assert j<i report "chybový stav";
assert clk='1' report "hodiny nejsou =1" severity WARNING;
```

Různé stupně důležitosti jsou označené slovy NOTE, WARNING, ERROR, FAILURE. Bez uvedení stupně se uvažuje stupeň ERROR.

### report

Vypíše hlášení.

[ návěstí: ] **report** string [ **severity** důležitost ] ;

```
report "Fáze 1 skončena";  -- důležitost NOTE
report "Nekonzistentní stav" severity FAILURE;
```

### přiřazení signálu <=

Přiřazení signálu lze považovat spíš za *konkurenční* příkaz než za sekvenční. Může být použit v sekvenčním kódu, ale takové použití s sebou nese vedlejší efekty, především změnu hodnoty až při ukončení sekvenčního kódu atd.

Signál navíc nelze definovat v sekvenčním kódu; musíte jej vždy definovat v nějaké nadřazené entitě.

[ návěstí: ] signál <= [ zpoždění ] průběh_signálu ;

kde:

*zpoždění* může být:

**transport**

**reject** čas

**inertial**

a *průběh_signálu* je:

prvek_průběhu [, prvek_průběhu]

**unaffected**

*Prvek_průběhu* pak je:

hodnota [ **after** čas ]

**null** [ **after** čas ]

Příklady:

```
sig1 <= sig2;
Sig <= Sa and Sb or Sc nand Sd nor Se xor Sf xnor Sg;
sig1 <= sig2 after 10 ns;
clk <= '1' , '0' after TimePeriod/2 ;
sig3 <= transport sig4 after 3 ns;
sig4 <= reject 2 ns sig5 after 3 ns;
sig6 <= inertial '1' after 2 ns, '0' after 3 ns , '1' after 7 ns;
```

Pokud neuvedete klauzuli **after**, je to totéž, jako byste napsali „after 0 fs;“ – tedy okamžitě.

Zpoždění „transport“ simuluje běžné zpoždění při vedení signálu. Zpoždění „inertial“ simuluje zpracování takovým obvodem, který vyžaduje, aby impuls měl nějakou minimální délku, než na něj zareaguje. Klauzule „reject“ říká, jaký musí být minimálně odstup od předchozího signálu. Pomocí těchto klauzulí můžete simulovat různé fyzikální jevy, jako parazitní kapacity apod.

### přiřazení proměnných :=

Přiřazuje hodnotu výrazu do proměnné.

[ návěstí: ] proměnná := výraz ;

```
A := -B + C * D / E mod F rem G abs H;
Sig := Sa and Sb or Sc nand Sd nor Se xor Sf xnor Sg;
```

### volání procedury

Volání procedury jejím jménem.

[ návěstí: ] jméno_procedury [ ( parametry ) ] ;

```
mojeProcedura;  -- bez parametrů
vypocet(parametr1, parA=>a, parB=>c+d);
```

Parametry mohou být přiřazené podle pozice, nebo v libovolném pořadí, pokud je pojmenujete.

### if

Podmíněná konstrukce.

```
[ návěstí: ] if  podmínka1  then
                  sekvence
             elsif  podmínka2  then   \_ nepovinné
                  sekvence            /
             elsif  podmínka3  then   \_ nepovinné
                  sekvence            /
             ...

             else                     \_ nepovinné
                  sekvence            /
             end if [ návěstí ] ;
```

```
if a=b then
    c:=a;
elsif b<c then
    d:=b;
    b:=c;
else
    necoDelej;
end if;
```

### case

Větvení příkazů podle možných hodnot výrazu.

```
[ návěstí: ] case výraz is
               when hodnota1 =>
                  sekvence
               when hodnota2 => \_ nepovinné
                  sekvence      /
               ...

               when others =>   \_ nepovinné
                  sekvence      /
             end case [ label ] ;
```

```
 case  my_val  is
```

```
    when 1 =>
      a:=b;
    when 3 =>
      c:=d;
      do_it;
    when others =>
      null;
  end case;
```

Příkazy ve větvi „when others“ se provedou v případě, že žádná z možností nevyhovovala výrazu. Pokud pokryjete všechny možnosti, není nutné tuto sekci uvádět.

### loop

Tři různé příkazy cyklu.

```
[ návěstí: ] loop
       sekvence -- nekonečná smyčka
                -- ukončíte příkazem exit
       end loop [ návěstí ] ;


[ návěstí: ] for proměnná in rozsah loop
                sekvence
           end loop [ návěstí ] ;


[ návěstí: ] while  podmínka loop
                sekvence
           end loop [ návěstí ] ;
```

Všechny cykly mohou obsahovat příkazy *next* a *exit* – ekvivalenty continue a break.

### next

Příkaz způsobí ukončení aktuálního cyklu a vyvolání další iterace – obdoba příkazu „continue“ v jazyce C s možností určení, který nadřízený cyklus se má iterovat dál.

[ návěstí: ] **next** [ návěstí2 ] [ **when** podmínka ] ;

```
next;
next vnejsi_cyklus;
next when A>B;
next this_loop when C=D or done; -- done je Boolean
```

### exit

Příkaz ukončí aktuální cyklus a skočí za jeho konec – je tedy ekvivalentní příkazu „break“ z jazyka C, opět s možností určení, který vnější cyklus se má ukončit.

[ návěstí: ] exit [ návěstí2 ] [ when podmínka ] ;

```
exit;
exit vnejsi_cyklus;
exit when A>B;
exit this_loop when C=D or done; -- done je Boolean
```

### return

Ukončení procedury nebo funkce. Ve funkci je povinný, v proceduře není nutný.

[ návěstí: ] **return** [ výraz ] ;

```
return; -- návrat z procedury (bez hodnoty)
return a+b; -- vracení hodnoty ve funkci
```

### null

Použijete v případech, kdy musíte použít nějaký příkaz, ale přitom není třeba nic dělat.

[ návěstí: ] **null** ;

## 20.6 Konkurenční příkazy

Konkurenční příkazy popisují architekturu, tedy „zapojení obvodu“. Hlavní rozdíl proti sekvenčním příkazům je ten, že konkurenční příkazy se dějí „najednou“, zatímco sekvenční se vykonávají jeden po druhém. Nezáleží tedy na pořadí, v jakém jsou zapsány, stejně jako nezáleží na tom, v jakém pořadí zapojujeme vodiče při stavbě reálné konstrukce – na funkci to většinou nebude mít vliv.

### block

Příkaz pro sloučení více konkurenčních příkazů do jednoho logického celku.

návěstí : **block** [ ( guard_signál ) ] [ **is** ]

[ generic [ generic map ; ] ]

[ port [ port map ; ] ]

[ deklarace ]

**begin**

konkurenční příkazy

**end block** [ návěstí ] ;

```
clump : block
        begin
          A <= B or C;
          D <= B and not C;
        end block clump ;

maybe : block ( B'stable(5 ns) ) is
           port (A, B, C : inout std_logic );
           port map ( A => S1, B => S2, C => outp );
           constant delay: time := 2 ns;
           signal temp: std_logic;
        begin
           temp <= A xor B after delay;
           C <= temp nor B;
        end block maybe;
```

### process

Proces slouží k provedení sekvenčních příkazů v konkurenčním prostředí.

návěstí : **process** [ ( sensitivity_list ) ] [ **is** ]

[ deklarace ]

**begin**

sekvenční příkazy

**end process** [ návěstí ] ;

```
reg_32: process(clk, clear)
        begin
          if clear='1' then
            output <= (others=>'0');
          elsif clk='1' then
            output <= input after 250 ps;
          end if;
        end process reg_32;

printout:  process(clk)
             variable my_line : LINE;
           begin
             if clk='1' then
               write(my_line, string'("at clock "));
               write(my_line, counter);
               write(my_line, string'("  PC="));
               write(my_line, IF_PC);
               writeline(output, my_line);
               counter <= counter+1;
             end if;
           end process printout;
```

V bloku deklarací mohou být deklarovány podprogramy, typy a subtypy, konstanty, proměnné, soubory, aliasy, atributy nebo skupiny, ale **ne signály**! Signály musí být deklarovány mimo proces.

Proces může být označený jako **postponed**, tedy odložený. Takový proces se vykoná ve stejném cyklu jako ostatní, ale až poté, co všechny *neodložené* procesy skončí.

### konkurenční volání procedury

Volání procedury je funkčně ekvivalentní tomu, jako kdyby na daném místě byl zapsaný příslušný proces.

[ návěstí : ] [ **postponed** ] jméno_procedury [ ( parametry ) ] ;

```
trigger_some_event ;

Check_Timing(min_time, max_time, clk, sig_to_test);
```

Procedura může být definována i v externí knihovně, na rozdíl od procesu.

### konkurenční assert

V konkurenčních blocích můžete použít i assert. Jeho chování bude stejné, jako kdyby byl zapsaný v procesu.

[ návěstí : ] [ **postponed** ] assert ;

### konkurenční přiřazení signálu

Přiřazení signálů je ekvivalent fyzického propojení pomocí vodičů. Přiřazení může být odložené (**postponed**) nebo hlídané (**guarded**).

```
[ návěstí : ] přiřazení;

[ návěstí : ] [ postponed ] podmíněné_přiřazení ;

[ návěstí : ] [ postponed ] výběrové_přiřazení ;
```

Volitelné hlídání (klauzule **guarded**) způsobí, že příkaz je proveden ve chvíli, kdy se hlídací signál změní z False na True.

### podmíněné přiřazení signálů

Podmíněné přiřazení, pokud je splněná podmínka.

signál <= průběh_signálu **when** podmínka;

signál <= průběh_signálu1 **when** podmínka **else** průběh_signálu2;

```
sig <= a_sig when count>7;
sig2 <= not a_sig after 1 ns when ctl='1' else b_sig;
```

Parametr „průběh_signálu“ jsme diskutovali už v příslušné sekci u sekvenčních příkazů.

### výběrové přiřazení signálů

Přiřazuje signálu jednu z několika různých hodnot, a to v závislosti na řídicím výrazu.

**with** výraz **select** signál <=

průběh_signálu **when** podmínka [, průběh_signálu **when** podmínka] ;

```
with count/2 select my_ctrl <=
               '1' when 1,
               '0' when 2,
               'X' when others;
```

### vytvoření komponenty

Vytvoří instanci zadané entity, popřípadě i konkrétní architektury.

jméno_instance: **entity** knihovna.jméno_entity[(jméno_architektury)]

**port map** ( připojení portů ) ;

jméno_instance: jméno_komponenty

**port map** ( připojení portů ) ;

### generate statement

Vytvoří více kopií konkurenčních příkazů (nejčastěji instancí nebo přiřazení).

```
návěstí: for proměnná in rozsah generate
    -- návěstí je povinné
      [ deklarace ]
   begin
     konkurenční příkazy  -- lze použít proměnnou
end generate návěstí ;

návěstí: if podmínka generate – návěstí je povinné
         [ deklarace ]
         begin
            konkurenční příkazy
         end generate návěstí ;
```

```
band : for I in 1 to 10 generate
b2 :    for J in 1 to 11 generate
b3 :     if abs(I-J)<2 generate
         part: foo port map ( a(I), b(2*J-1), c(I, J) );
         end generate b3;
        end generate b2;
       end generate band;
```

**DATA, ČIPY, PROCESORY**
**Vlastní integrované obvody na koleni**

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2020
Kniha vyšla jako 25. publikace v Edici CZ.NIC.
Tisk: H.R.G. spol. s r.o., Svitavská 1203, 570 01, Litomyšl
Sazba: Karel Slanař



ISBN 978-80-88168-53-9 (tištěná verze)
ISBN 978-80-88168-54-6 (ve formátu EPUB)
ISBN 978-80-88168-55-3 (ve formátu MOBI)
ISBN 978-80-88168-56-0 (ve formátu PDF)

Edice CZ.NIC

**O knize** Mikroprocesory přinesly obrovský pokrok v elektronice, ale ta se nezastavila. Dnešními nejdiskutovanějšími obvody jsou programovatelná logická pole (FPGA), která v sobě integrují obrovské množství logických prvků. To, co je na nich nejlákavější, je fakt, že jejich vnitřní zapojení určuje zákazník – tedy vy. Z logického pole se tak může stát celý elektronický systém, poskládaný do posledního prvku tak, jak si přejete. Chcete zařízení pro paralelní zpracování dat? Několik procesorů na jednom čipu? Neuronovou síť pro počítačové vidění? Nebo výkonný bitcoin miner? Nebo paralelní procesor pro 3D grafiku? Nebo celý počítač? Nebo snad vlastní výkonný síťový router? Není problém! Vše na míru, račte si vybrat!

Autor ve své třetí knize volně navazuje na předchozí tituly a po úvodu do elektroniky a základech mikroprocesorových konstrukcí přináší úvod do práce s programovatelnými logickými poli. Podrobně se věnuje výkladu jazyka VHDL, který slouží k popisu logických obvodů, a od základních principů pokračuje přes složitější obvody a komponenty až k „systémům na jednom čipu". Na konci výkladu si díky nabytým znalostem zkonstruujete společně vlastní šestnáctibitový mikroprocesor i s jednoduchým programovým vybavením. Každý krok a každou naučenou dovednost si rovnou vyzkoušíte a aplikujete v praktických příkladech.

**O autorovi** **Martin Malý** (známý též pod přezdívkou Arthur Dent či Adent), je český programátor, bloger, publicista a komentátor. V roce 2003 naprogramoval a spustil veřejnou blogovací službu Bloguje.cz, od roku 2007 psal pravidelný sloupek pro Digiweb – součást webu iHNed.cz. Později pracoval na pozici šéfredaktora webového magazínu Zdroják (vydávala společnost Internet Info, s.r.o.). V březnu 2013 nastoupil do vydavatelství Economia na pozici vedoucího týmu redakčních vývojářů iHNed.cz. Od roku 2015 popularizuje Internet věcí a DIY elektroniku, přednáší o těchto oborech a školí zájemce o platformu Arduino. Za svou popularizační činnost byl v roce 2016 nominován v anketě Křišťálová Lupa, v kategorii One (wo)man show. Od roku 2017 je zástupcem šéfredaktora iHNed.cz pro rozvoj a placený obsah. Elektronika je jeho koníčkem už od dětství. Nejraději má staré osmibitové počítače z osmdesátých let 20. století – vlastní jich několik desítek a stále je jimi fascinován natolik, že si staví jejich repliky a programuje jejich emulátory. Ve spolupráci se sdružením CZ.NIC přednáší v kurzu „Arduino pro učitele".

**O edici** Edice CZ.NIC je jednou z osvětových aktivit správce české národní domény. Ediční program je zaměřen na vydávání odborných, ale i populárně naučných publikací spojených s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz.

knihy.nic.cz

ISBN 978-80-88168-56-0

9 788088 168560