Návrh aritmetických operátorů na FPGA

Zatímco předchozí díly seriálu byly zaměřeny spíše na logický a systémový návrh obecných obvodů, v tomto dílu se budeme věnovat návrhu speciálních struktur určených pro aritmetické výpočty na programovatelných hradlových polích. V příspěvku představíme možnosti realizace základních aritmetických operátorů (inverze znaménka, výpočtu absolutní hodnoty, sčítání/odčítání, násobení a dělení) s ohledem na reálnou implementaci na FPGA obvodech. Text je opět doplněn příklady VHDL kódu. Krátce se zmíníme i o vlastních aritmetických knihovnách v jazyce VHDL, jejich použití a účelu.

POUŽITÉ ZKRATKY

BS	*Barrel Shifter*
LSB	*Least Significant Bit*
LUT	*Look Up Table*
MSB	*Most Significant Bit*
RTL	*Register Transfer Level*
STA	*Statical Timing Analysis*
WE	*Write Enable*

1 Úvod

Moderní FPGA obvody jsou velmi často používány pro aplikace v oblasti číslicového zpracování signálu a jejich architektury jsou tomu proto patřičně přizpůsobeny. Podobně i nástroje pro návrh obvodů na programovatelných hradlových polích poskytují značnou podporu pro návrh aritmetických operátorů, a návrháři tak velmi zjednodušují práci. Jejich implementace nicméně nemusí být vždy bezproblémová; proto je tento příspěvek věnován základům realizace aritmetických operátorů na obvodech FPGA. V příspěvku nejprve krátce zopakujeme problematiku zobrazení čísel v počítači. Dále se budeme věnovat po řadě detailům realizace invertoru znaménka, sčítačky a odečítačky, násobičky a děličky se zvláštním zřetelem k jejich implementaci na programovatelných hradlových polích. Jako u předchozích článků je i tento text doplněn příklady popisovaných struktur implementovaných v VHDL, kódy lze stáhnout z adresy [1].

2 Zobrazení čísel

Se základním formátem pro zobrazení kladných čísel – přímým kódem – se jistě už všichni čtenáři setkali. Jako příklad může sloužit zobrazení čísel 60 a 196 pomocí osmibitového slova

$60d = 00111100b = 0\times2^7+0\times2^6+1\times2^5+1\times2^4+$
$+1\times2^3+1\times2^2+0\times2^1+0\times2^0,$
$196d = 11000100b = 1\times2^7+1\times2^6+0\times2^5+$
$+0\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0.$

Stejně jako v desítkové soustavě má každá dvojková číslice přiřazenu odpovídající váhu. Protože zobrazujeme pouze nezáporná čísla, lze pomocí N bitového čísla zobrazit dekadická čísla z rozsahu 0 … + $+2^N - 1$.

Pro zobrazení celých čísel je běžně používán doplňkový kód (*two's complement* – dvojkový doplněk). Jediná odlišnost doplňkového od přímého kódu tkví ve váze nejvyššího bitu; podívejme se jaká dekadická čísla dostaneme pro příslušná osmibitová slova uvedená výše

$00111100b = 0\times(\mathbf{-2^7})+0\times2^6+1\times2^5+1\times2^4+$
$+1\times2^3+1\times2^2+0\times2^1+0\times2^0 = +60d,$
$11000100b = 1\times(\mathbf{-2^7})+1\times2^6+0\times2^5+0\times2^4+$
$+0\times2^3+1\times2^2+0\times2^1+0\times2^0 = -60d.$

Všimněme si, že nejvyšší bit má váhu -2^{N-1} pro N bitové číslo. Není to tedy znaménkový bit, jak se někdy mylně označuje. Uvědomíme-li si tuto skutečnost, řada aritmetických operací v doplňkovém kódu se rázem stane srozumitelnější. Rozsah zobrazitelných čísel je zde $-2^{N-1} \ldots + 2^{N-1} - 1$.

Doposud jsme hovořili pouze o celých číslech. Z pohledu implementace aritmetických operací většinou příliš nezáleží na tom, kde je umístěna řádová čárka. V případech uvedených výše jsme nejnižšímu řádu přiřadili váhu 2^0. Při práci s čísly v oblasti číslicového zpracování signálu bývá nicméně běžné umisťovat řádovou čárku těsně před nejvyšší bit; získáme tak tzv. zlomkový (*fraction*) formát. Výše uvedené příklady pak budou vypadat následovně (poloha řádové číslice před čárkou je úmyslně zvýrazněna):

$\mathbf{0},0111100b = 0\times(\mathbf{-2^0})+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}+$
$+1\times2^{-4}+1\times2^{-5}+0\times2^{-6}+0\times2^{-7} = +0{,}46875d,$
$\mathbf{1},1000100b = 1\times(\mathbf{-2^0})+1\times2^{-1}+0\times2^{-2}+0\times2^{-3}+$
$+0\times2^{-4}+1\times2^{-5}+0\times2^{-6}+0\times2^{-7} = -0{,}46875d$

Rozsah zobrazitelných čísel je v tomto případě $-1 \ldots +1 - 2^{-N+1}$.

V technické praxi se lze setkat i s dalšími kódy – s posunutou nulou, inverzním kódem, jednotkovými doplňkem atd. [2]. Samostatnou kapitolou je pak zobrazení čísel v pohyblivé řádové čárce [2].

5 Základní aritmetické operace

5.1 Inverze znaménka

Inverze znaménka je v mnoha případech užitečná operace. Otočením znaménka operandu můžeme například převést sčítání na odečítání či určit absolutní hodnotu čísla. Inverze znaménka čísla a v doplňkovém kódu je prováděna podle známého pravidla; z vlastností doplňkového kódu plyne, že

$$-a = 2^N + a = (2^N-1) - a + 1 = \mathrm{NOT}(a) + 1$$

Chceme-li tedy invertovat znaménko čísla, invertujeme všechny jeho bity a přičteme jedničku. Tomu odpovídá i schéma na *obr. 1*; červeně je označena část sloužící k vlastní operaci otočení znaménka, celé schéma představuje jednotku pro výpočet absolutní hodnoty. Protože rozsah čísel zobrazitelných ve dvojkovém doplňku je asymetrický a hodnota -2^{N-1} nemá kladný protějšek (největší zobrazitelné kladné číslo je $2^{N-1} - 1$), je třeba výstup obvodu chápat jako N bitové číslo v přímém kódu a nejvyšší bit „nezahazovat“. Alternativou je obvodové řešení mírně zkomplikovat a přidat detekci čísla -2^{N-1} na vstupu. Výstup potom bude dodatečnou logikou saturován do $2^{N-1} - 1$. Aplikace ovšem musí takový postup umožňovat. Implementace schématu z *obr. 1* v jazyce VHDL je triviální:

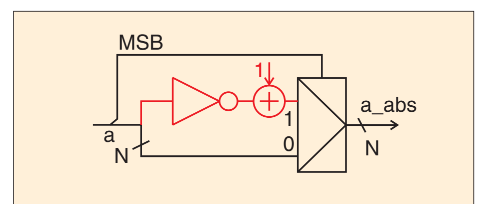
```
a_abs <= a WHEN a(N–1) = ‘0’ ELSE NOT(a)
+ 1;
```

Všimněte si jednoduché implementace přičtení 1 pomocí operátoru „+“. Aby bylo možné takto jednoduše implementovat sčítačku, je nejprve nutné zavést knihovny pro aritmetické operace použitím klauzule USE:

```
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.std_logic_signed.ALL;
```

Začátečníkovi může systém aritmetických knihoven v jazyce VHDL připadat poněkud nepřehledný, uveďme proto stručnou informaci o tom, k čemu jsou jednotlivé knihovny dobré:

- std_logic_unsigned: knihovna obsahuje syntetizovatelné aritmetické operátory (+, –, *, ABS, relační operátory), které pracují s typem std_logic_vector. Knihovnu připojte pokud budete chtít pracovat jen s čísly bez znaménka.
- std_logic_signed: totéž, jako std_logic _unsigned, ale pro čísla se znaménkem,



Obr. 1 Obvod pro určení absolutní hodnoty (část určená k inverzi znaménka je zvýrazněna červeně)

- std_logic_arith: tuto knihovnu většinou užijeme tehdy, chceme-li současně pracovat s čísly s a bez znaménka. Knihovna zavádí nové datové typy signed a unsigned odvozené z typu std_logic_vector. Jejich použitím říkáme, že binární číslo na příslušné sběrnici chápeme jako číslo bez znaménka (unsigned), nebo ve dvojkovém doplňku (signed). Protože jsou oba typy odvozené z typu std_logic_vector, pracuje se s nimi stejným způsobem. Knihovna dále obsahuje několik užitečných funkcí pro konverzi mezi typy std_logic_vector, unsigned, signed a integer.

Čtenáři zde doporučujeme důkladné prostudování zdrojových kódů těchto knihoven, lze je typicky najít v instalačním adresáři vašeho simulátoru. Například u ModelSimu XE je to adresář program files\Mxeiii\Xilinx\vhdl\src\ieee.

5.2 Sčítání a odečítání

Operace sčítání $c = a + b$ je využívána jak všemi složitějšími obvody (například každým čítačem ve vašem obvodu), tak i operátory násobení a dělení. *N*-bitová sčítačka je tak klíčovým aritmetickým operátorem a na její rychlosti a dalších vlastnostech závisí parametry celého systému pro číslicové zpracování signálu.

Běžně dostupná literatura o logickém návrhu, např. [2 a 3], vždy uvádí elementární odvození logických funkcí a zapojení jak poloviční (*half*), tak plné sčítačky (*full adder*). Dovolíme si proto předpokládat, že čtenář je s těmito konstrukcemi již důvěrně obeznámen a zaměříme se spíše na specifika návrhu na obvodech FPGA.

Nejjednodušší možná implementace paralelní sčítačky je tzv. ripple carry adder, sčítačka s „čeřením přenosu" (*obr. 2*). Návrh sčítačky v jazyce VHDL je opět triviální:

```
c <= a + b;
```

Tato implementace sčítačky je přímo použitelná jak pro sčítání čísel v přímém, tak v doplňkovém kódu.

Nevýhodou obvodu je jeho poměrně vysoké kombinační zpoždění způsobené šířením přenosu mezi jednotlivými řády sčítačky. Jeden možný případ šíření přenosu lze nalézt na *obr. 2*. Nejdelší možná kombinační cesta obvodem (kritická cesta) vede ze vstupu a_0, nebo b_0 na výstup cy_{N-1} (cy_7 v tomto případě) a celkové zpoždění závisí na šířce slova *N*. To může negativně ovlivnit maximální pracovní frekvenci f_{max} celého obvodu, např. [4].

Existuje mnoho způsobů, jak kombinační zpoždění sčítačky zkrátit [2 a 3], všechny ovšem vyžadují netriviální zásah do struktury obvodu. Drtivá většina dnešních obvodů FPGA tento problém řeší pomocí dedikovaného kanálu pro akceleraci přenosu (tzv. carry chain). Použijeme-li standardní způsob implementace sčítačky v jazyce VHDL uvedený výše, nástroje pro implementaci automaticky zajistí užití příslušných struktur pro zrychlení přenosu. Jestliže i přesto výsledná struktura kriticky snižuje f_{max}, je kromě použití sofistikovanějšího obvodového řešení možné problém řešit i pomocí proudového zpracování.

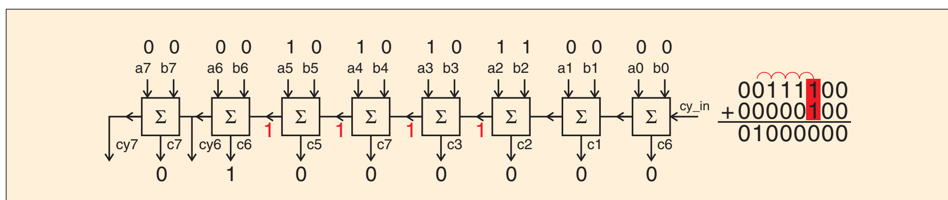
Dále je při sčítání nezbytné brát v úvahu možné přetečení. Sčítáme-li dvě *N*-bitová čísla, získáváme obecně $N + 1$-bitový výsledek. Je-li výstup sčítačky jen *N*-bitový, je třeba detekovat přetečení a případně provést následnou korekci výsledku. Obecně mohou nastat dva případy. Sčítáme-li pouze kladná *N*-bitová čísla v přímém kódu, je přetečení indikováno přenosem z nejvyššího bitu sčítačky (*obr. 2*, výstup cy_7). Pracujeme-li s čísly v doplňkovém kódu, přetečení může nastat jen tehdy, pokud sčítáme čísla se stejným znaménkem. Přetečení pak rozpoznáme tak, že výsledek je opačného znaménka, než oba operandy ($a_7 = b_7$, $c_7 \neq a_7$). Druhou možností detekce je srovnat přenosy z a do nejvyššího řádu ve sčítačce. Jsou-li oba shodné (v *obr. 2* $cy_6 = cy_7$), k přetečení nedošlo. V případě detekce přetečení můžeme dále výsledek ošetřit pomocí saturace – místo poškozeného výsledku užijeme největší zobrazitelné číslo s očekávaným znaménkem původního výsledku. To je charakteristické pro aplikace v oblasti číslicového zpracování signálů. Sčítání se saturací pak lze provést pomocí následujícího kódu VHDL (příklad je pro $N = 8$ z důvodů názornosti a přehlednosti zápisu):

```
c_sat_add_i <= a + b;
overflow_i <= '1' WHEN a(7) = b(7) AND
c_sat_add_i(7) /= a(7) ELSE '0';
c_sat_add <= "01111111" WHEN overflow_i
= '1' AND a(7) = '0' ELSE
 "10000000" WHEN overflow_i = '1' AND
a(7) = '1' ELSE
                c_sat_add_i;
```

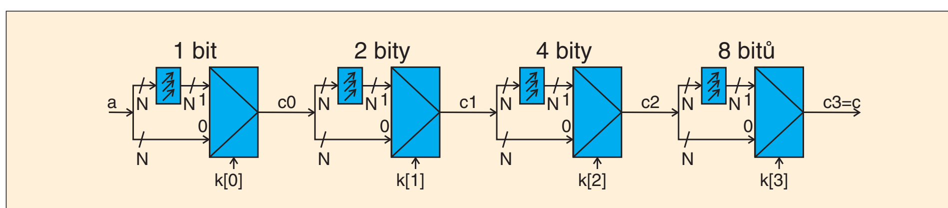
Jestliže sčítáme dvě *N*-bitová čísla v doplňkovém kódu a chceme získat $N + 1$-bitový výsledek, je před sčítáním nezbytné provést znaménkové rozšíření (*sign extension*) obou sčítanců na $N + 1$ bitů. To provedeme jednoduchou kopií nejvyššího bitu:

```
c <= (a(N – 1)&a) + (b(N – 1)&b);
```

Chceme-li místo sčítání odečítat, lze jednoduše použít VHDL operátor „–", případně sčítačku zkombinovat s obvodem pro inverzi znaménka vyžadujeme-li přepínat mezi sčítáním a odečítáním za běhu.



Obr. 2 Paralelní sčítačka se zvýrazněnou kombinační cestou – přenos



Obr. 3 Barrel shifter pro posuv čísla až o 15 bitů – násobení 2^0, 2^1, ..., 2^{15}

Tab. 1 Příklad funkce barrel shifter – hodnoty na interních signálech

k[0] = 1	*c0* = 2 × *a*
k[1] = 0	*c1* = *c0* = 2 × *a*
k[2] = 0	*c2* = *c1* = 2 × *a*
k[3] = 1	*c3* = 256 × *c2* = 512 × *a* = 2^9 × *a*

Na závěr krátké diskuze možností implementace sčítačky poznamenejme, že nejčastěji v praxi sčítáme proměnný operand *a* s konstantním operandem *b* – například při inkrementaci hodnot v čítači. V takovém případě místo operandu *b* ve VHDL kódu jednoduše použijeme příslušnou konstantu. Nástroje pro syntézu a implementaci toto automaticky rozpoznají a sčítačku optimalizují vzhledem k použité konstantě. Ve výsledném obvodu potom nebude plná sčítačka, ale specializovaný obvod například pro přičtení jedničky. Čtenáře zde odkazujeme i na minulé díly našeho seriálu, kde jsme čítače i sčítačku již intuitivně použili.

5.3 Násobení

Násobení je v oblasti číslicového zpracování signálu jednou z nejdůležitějších operací (hned po sčítání). Tomu odpovídá i široká škála dostupných řešení a postupů, které pro něj byly vyvinuty.

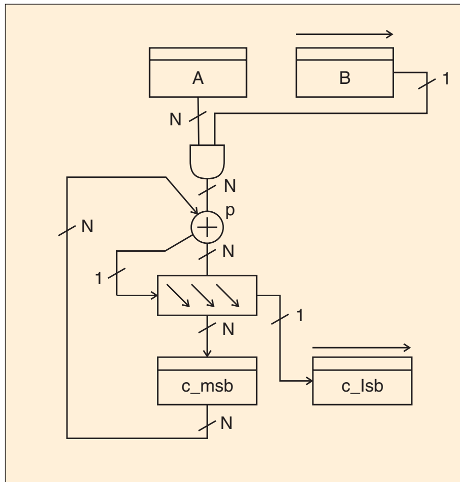
Předpokládejme, že násobíme dvě celá čísla *a*, *b*; získáme výsledek $c = a.b$. Jsou-li činitelé *a* a *b* *N*-bitová čísla, musí být součin *c* v nejhorším případě 2*N* bitů široký, aby nedošlo k přetečení.

Technicky nejjednodušším případem je násobení proměnného činitele *a* konstantní přirozenou mocninou dvou, tedy $b = 2^k$. Násobení je pak redukováno na pouhý bitový posuv, v jazyce VHDL ho lze implementovat takto:

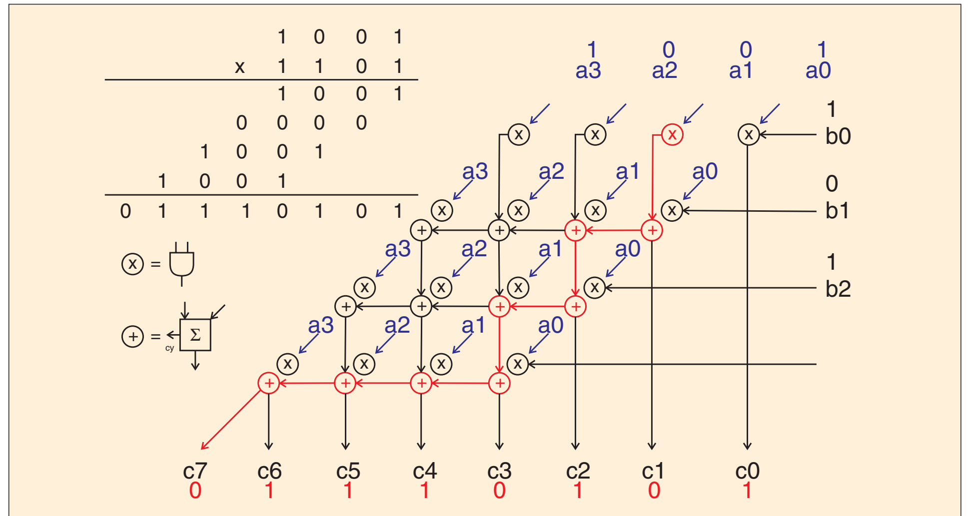
```
c(N+k-1 DOWNTO k) <= a; c(k-1 DOWNTO 0)
<= (OTHERS => '0');
```

Zde ani nezáleží na tom, zda činitel úplnost zmiňme, že bezesporu nejjednodušším řešením je postupné sčítání (operand *a* přičteme *b* krát k sama sobě). To ovšem pro extrémní pomalost téměř nikdy nepoužijeme.

Dříve velmi populárním řešením bylo použití sériové násobičky. Její nespornou výhodou je velikost – násobička zabírá minimum plochy na obvodu FPGA. Nevýhodami jsou pomalé násobení (pro *N*-bitové činitele je třeba *N* hodinových cyklů) a potřeba složitějšího řízení celého obvodu. Na *obr. 4* lze nalézt její schéma, jedná se tentokrát o sekvenční obvod. Násobení probíhá podle dobře známého „ručního" algoritmu; v *tab. 2* je uveden příklad postupu při násobení spolu s obsahy jednotlivých registrů. Činitele *a* a *b* na počátku násobení uložíme do registrů *A* a *B*. Registr *B* je posuvný, s každým hodinovým cyklem se posouvá jeho obsah vpravo a jeho nejnižší bit je použit pro výpočet dílčího součinu *p*. Výpočet postupuje po krocích, v *N*-tém kroku (hodinovém cyklu) získáme *N*-tý bit výsledku, protože tento závisí jen na mezivýsledcích z kroků 1, ..., *N*. Výsledný bit se uloží do registru *C_lsb* (opět posuvný registr, v každém kroku se jeho obsah posouvá o bit vpravo). Po *N* krocích je k dispozici výsledek ve dvojici registrů *C_msb* a *C_lsb*. Popsaný postup platí pro nezáporná čísla, jsou-li operandy se znaménkem, pak v posledním kroku dílčí součin *p* odečítáme, protože nejvyšší bit ve dvojkovém doplňku má váhu s negativním znaménkem.



Obr. 4 Zjednodušené schéma sériové násobičky (všechny registry mají šířku *N* bitů)



Obr. 5 Paralelní násobička s příkladem násobení

a a součin *c* jsou v přímém kódu, nebo v dvojkovém doplňku. Je-li *b* požadováno záporné, lze po posuvu zařadit už popsaný obvod pro inverzi znaménka. Poněkud zajímavější případ nastává pokud požadujeme mocninu *k* proměnnou v době funkce obvodu. Pak je na místě použít obvod označovaný jako *barrel shifter* (BS), schéma je na *obr. 3*. Obvod funguje velmi jednoduše, jeho funkce je postavená na rozkladu binárního čísla na řády. Představme si například, že chceme vstup *a* vynásobit 2^9, $k = 9\text{d} = 1001\text{b}$. Potom budou na jednotlivých interních signálech obvodu hodnoty jako v *tab. 2*.

VHDL kód BS je jen jednoduchou kombinací multiplexerů a posuvu vlevo. Zajímavou techniku pro implementaci obecného BS prezentuje dokument [5]. Poznamenejme zde ještě, že *barrel shifter* se často implementuje pomocí bitových rotací, nikoliv posunů. V tom případě ho ale nelze jednoduše použít pro násobení; zde je nezbytné vkládat zprava vždy nuly.

V případě obecného operandu *b* máme k dispozici široké spektrum řešení od plně sekvenční až po plně paralelní násobičku. Pro

Tab. 2 Příklad sériového násobení (nejnižší bit registru *B* určující součin *p* je zvýrazněn)

krok	*B*	*p*	*C_msb*	*C_lsb*
1	110**1**	1001	0100	1000
2	011**0**	0000	0010	0100
3	001**1**	1001	0101	1010
4	000**1**	1001	0111	0101

Dnes sériovou násobičku na FPGA používáme spíše ve výjimečných případech. Řešení, které odstraňuje nevýhodu nízké rychlosti výpočtu za cenu zvětšení plochy obvodu a zpoždění logiky, představuje paralelní násobička. Tu si lze v nejjednodušším případě představit jako sériovou násobičku „rozbalenou do prostoru" (*obr. 5*). Násobička je nyní jeden veliký kombinační obvod. Ve schématu je červenou barvou zdůrazněná kritická cesta strukturou.

Nástroje pro syntézu a implementaci obvodů na FPGA návrháři při implementaci paralelní násobičky ušetří mnoho práce. Vše, co musíme udělat, je zapsat v jazyce VHDL jednoduchou konstrukci. Následující příkazy ukazují implementace násobičky jak pro čísla se znaménkem, tak pro čísla bez znaménka:

```
—nasobeni bez znamenka
c_us_mult_i <= unsigned(a) * unsigned(b);
c_us_mult <= std_logic_vector(c_us_mult_i);
```

```
—nasobeni se znamenkem
c_s_mult_i <= signed(a) * signed(b);
c_s_mult <= std_logic_vector(c_s_mult_i);
```

kde a, b, c_s_mult, c_us_mult jsou typu std_logic_vector. Popsaný zápis vyžaduje importovat knihovnu std_logic_arith.

Výsledná implementace násobičky velmi závisí na rodině použitého obvodu FPGA. Při použití starších či levnějších obvodů FPGA je vygenerovaná struktura násobičky složená z diskrétních logických prvků (např. LUT). Taková násobička bude patrně zabírat nezanedbatelnou část plochy obvodu FPGA. Naproti tomu moderní programovatelná hradlová pole mají integrované hardwarové násobičky a nástroje pro syntézu, rozmístění a propojení je dokáží využít. Výhodami integrovaných násobiček je nižší zpoždění, menší zabraná plocha obvodu FPGA a ovšem i celkově nižší příkon obvodu FPGA při násobení. Například v případě FPGA řady Virtex se můžeme setkat s integrovanými násobičkami násobícími operandy 18 × 18 bitů se znaménkem, výsledek má šířku 36 bitů se znaménkem (případně 17 × 17 bitů bez znaménka s 34 bitů širokým výsledkem). Nástroje pro implementaci násobičku automaticky použijí. Navíc násobíme-li slova o jiné šířce, než je 18 bitů se znaménkem, je automaticky zajištěna konverze na nativní šířku. Násobička pak buď není zcela využita (jsou-li operandy užší, než 18 bitů), nebo nástroj použije více integrovaných násobiček (jsou-li operandy širší), případně i v kombinaci s další logikou. Práce

s integrovanými násobičkami nicméně nemusí být zcela zdarma, neboť například na FPGA Virtex násobičky sdílí propojovací kanály s bloky blokové paměti (BRAM). Použití jedné násobičky tak zabírá i jeden blok BRAM.

V technické praxi se může snadno stát, že násobičkou prochází kritická cesta a tedy zpoždění logiky pro násobení významně snižuje maximální hodinovou pracovní frekvenci obvodu. Významnou pomocí v tomto případě může být implementace násobičky s proudovým zpracováním – *pipelined multiplier* [6].

5.4 Dělení

Operátor dělení je nejméně často implementovaným základním aritmetickým operátorem. Dělení $c = a/b$ je samo o sobě celkem pomalou operací, sekvenčním výpočtem na několik period hodin, a proto se ho obvykle snažíme nahradit alternativními implementacemi (nejčastěji převodem na násobení).

Potřebujeme-li implementovat operaci dělení, může opět nastat několik různých situací podle požadavků na operátor. Probereme je v pořadí podle vzrůstající obtížnosti. Nejjednodušším případem je znovu dělení celočíselnou konstantní mocninou dvojky, $b = 2^k$. Opět použijeme jednoduchý bitový posuv, tentokrát vpravo

```
c(N-1 DOWNTO N-k) <= (OTHERS => ‘0’);c(N-k-1 DOWNTO 0) <= a(N-1 DOWNTO k);
```

Jsou-li operandy čísla se znaménkem, situace se mírně zkomplikuje. Při bitovém posunu vpravo je nutno správně pracovat s nejvyšším bitem a v principu zleva nasouvat kopii znaménka *(sign extension* – znaménkové rozšíření):

```
c(N-1 DOWNTO N-k) <= (OTHERS => a(N-1));c(N-k-1 DOWNTO 0)<=a(N-1 DOWNTO k);
```

Navíc je nutné si uvědomit existenci malé komplikace spojené s doplňkovým kódem. Představme si, že dělíme dvěma – tedy posouváme o bit vpravo – následující čísla:

+1d/2 = 00000001b/2 = 00000000b = 0d,
–1d/2 = 11111111b/2 = 11111111b = –1d
+8d/2 = 00001000b/2 = 00000100b = 4d,
–8d/2 = 11111000b/2 = 11111100b = –4d
+5d/2 = 00000101b/2 = 00000010b = 2d,
–5d/2 = 11111011b/2 = 11111101b = –3d

Vidíme, že výsledek je vždy „zaokrouhlení“ směrem k minus nekonečnu. To u kladných čísel nečiní problém, jelikož vypočtený podíl je v absolutní hodnotě vždy menší, nebo roven, než jeho skutečná hodnota. U záporných čísel je ale vypočtený podíl v absolutní hodnotě vždy větší, nebo roven skutečné hodnotě. Extrémní případ z hlediska relativní chyby nastává dělíme-li –1 libovolnou nezápornou mocninou dvou – dostáváme stále –1. S tímto jevem je třeba počítat, a je-li na závadu, operaci patřičně přizpůsobit. Nejjednodušší postup je provést nejprve výpočet absolutní hodnoty, posléze vlastní posuv, a nakonec obnovit původní znaménko případným násobením –1.

Požadujeme-li k proměnné za běhu aplikace, užijeme obvod z *obr. 3*, jen posuv vlevo nahradíme posuvem vpravo.

Při implementaci nejrůznějších algoritmů často potřebujeme dělit obecnou konstantou b, jež ovšem není mocninou dvojky. Dělení v tomto případě provedeme pomocí násobení převrácenou hodnotou b.

Konečně, nejsložitější případ nastává, má-li být b obecné a proměnné za běhu obvodu. Nejjednodušší a v praxi v podstatě nepoužívané řešení je dělení postupným odečítáním

$$p_0 = a,\ p_{i+1} = p_i - b$$

tak dlouho, dokud je $p_{i+1} \le 0$. Potom $c = i$. Zásadní nevýhodou je zde extrémní pomalost výpočtu.

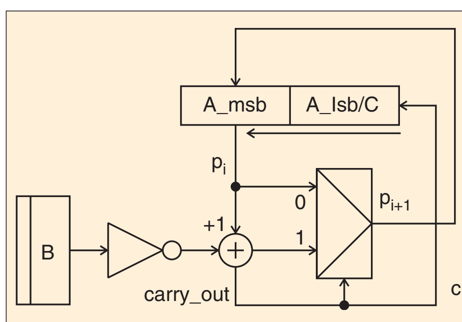
Elegantnější řešení představuje tzv. sériová dělička [7]. Sériová dělička v nejjednodušším případě provádí dělení podle dobře známého „ručního“ algoritmu dělení čísel. Algoritmus je iterativní, v každém kroku vypočte mezivýsledek p_i a novou číslici podílu c_i (v tomto případě jeden bit, základ je 2). Počáteční podmínka je $p_0 = a$. V každém kroku hledáme c_i a vypočteme nové p_{i+1}:

$$c_i = \text{floor}(p_i/b),\ p_{i+1} = 2(p_i - c_i b),$$

kde funkce floor(x) vrací celočíselnou část x.

Předpokládáme ovšem, že jak dělenec, tak dělitel je před dělením normalizován do rozsahu 2^N–1, ..., 2^{N-1}, aby v prvním kroku bylo c_1 nejvýše 1. To lze udělat násobením a i b pomocí vhodné mocniny dvojky. Ve skutečnosti tak budeme dělit $c' = (a2^N)/(b2^M)$. Pro výsledek platí $c = 2^{M-N}c'$.

Při implementaci tohoto algoritmu je nejnáročnějším krokem stanovení c_i. Jedno z možných řešení je provést odečtení $p_i - b$ a rozhodnout se podle výsledku. Finální implementace děličky pro nezáporné operandy pak může být jako na *obr. 6* [8]. Celý obvod funguje poměrně jednoduchým způsobem: na začátku dělení se do registrů A, B nahrají dělenec a a dělitel b. Registr A je v tomto zapojení posuvný, v každém kroku dělení se jeho obsah posouvá o bit vlevo. Zprava se do něj nasune nová číslice podílu c_i. Horní polovina registru *A_msb* je stejně široká, jako registr dělitele B. Během každého taktu dělení se ve sčítačce odečte *A_msb* – B, odečtení je zajištěno pomocí inverze B a přičtení jedničky do nejnižšího řádu sčítačky. Na základě přenosu z nejvyššího řádu sčítačky lze zjistit, zda *A_msb* >= B (pak *carry_out = 1*), či naopak (*carry_out = 0*). Význam hodnoty signálu *carry_out* je zde jen zdánlivě obrácený (1 pro kladný a 0 pro záporný výsledek). Souvisí to s tím, že pracujeme jen s kladnými čísly – čtenář nechť si zde rozmyslí důvod. Signálem *carry_out* konečně vybereme nový obsah registru *A_msb* a současně představuje signál nese novou číslici podílu (1 pro *A_msb* >= B, 0 jinak*)*.



Obr. 6 Zjednodušené schéma sériové děličky nezáporných čísel

Popsaný algoritmus a zapojení jsou velmi primitivní, ve skutečnosti existuje mnoho vylepšení sekvenční děličky, např. [7]. Vylepšený algoritmus (tzv. SRT algoritmus) sekvenčního dělení je použit například v procesorech řady Pentium.

Jinou zajímavou možností implementace dělení je dělení pomocí násobení. Operaci $c = a/b$ potom převedeme na $c = a(1/b)$, a vlastní násobení tak musí předcházet aproximace převrácené hodnoty dělitele. Tu lze určit jednoduchým iterativním algoritmem. Podrobný popis celého algoritmu je uveden v [7]. Podobný algoritmus je užit v moderních procesorech firmy AMD.

Matematické knihovny dostupné v jazyce VHDL neposkytují žádnou „zkratku“ pro implementaci obecné děličky. Návrhář v tomto případě musí sám implementovat schéma na *obr. 6*, nebo jiné podobné řešení.

7 Závěr

Problematika implementace aritmetických operací je velmi široká a v tomto příspěvku jsme představili jen nejzákladnější řešení triviálních operací. Čtenáři tak doporučujeme i další studium odborné literatury; dobrým úvodem do problematiky jsou například lehce dostupné publikace [2 a 3]. Mnoho zajímavých rad pro návrh aritmetických obvodů na FPGA lze najít i v dokumentech firmy Xilinx, např. [9].

Kompletní ukázkové kódy všech aritmetických operátorů i s jednoduchým VHDL testbenchem a simulačními skripty lze opět nalézt na [1].

Tato práce byla podpořena výzkumným záměrem MSM6840770012 – Transdisciplinární výzkum v biomedicínckém inženýrství 2.

Ing. Jakub Šťastný, Ph.D.
katedra teorie obvodů FEL ČVUT v Praze,
ASICentrum, s. r. o.

Literatura k čláínku je na str. 94