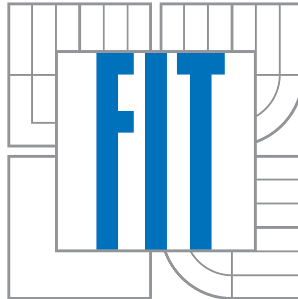
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Obvodová realizace vyvíjejících se systémů

Diplomová práce

2006 Zdeněk VAŠÍČEK

Obvodová realizace vyvíjejících se systémů

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně dne 24. května 2006

© ZdeněkVašíček, 2006

Autor práce tímto převádí svá práva na reprodukci a distribuci kopií celého díla i jeho částí na Vysoké učení technické v Brně, Fakultu informačních technologií.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Lukáše Sekaniny, Ph.D a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Zdeněk Vašíček
24. května 2006

Abstrakt

Tato práce se zabývá praktickou realizací vyvíjejícího se obvodu. Cílem je navrhnout a implementovat kompletně hardwarové řešení založené na rekonfigurovatelném hradlovém poli FPGA Virtex II Pro. Hlavní výhodou tohoto přístupu je vysoká rychlost. Evoluce má možnost prozkoumat ve stejném čase mnohem větší prostor možných řešení než v případě čistě softwarového řešení. Navržený vyvíjející se obvod demonstrujeme na problému evolučního návrhu obrazových operátorů. Úlohou evolučního algoritmu je nalézt filtr, který minimalizuje odchylku mezi požadovaným obrazem a odezvou získanou filtrací vstupního poškozeného obrazu. Jelikož již existuje kompletně hardwarové řešení, bude snahou vytvořit systém umožňující získat lepší výsledky. Právě proto jsme se rozhodli použít procesor PowerPC, který dovolí, narozdíl od stávajícího řešení, realizovat kvalitnější evoluční algoritmus. Časově náročnější fáze výpočtu fitness hodnoty běží v hardware a zbylá část evolučního algoritmu v software. Získané výsledky ukazují, že navržené řešení je schopno ve stejném čase produkovat kvalitnější obrazové operátory než existující řešení. Mimoto se podařilo navrhnout obecné obrazové operátory pro detekci hran a eliminaci impulsního šumu.

Klíčová slova

Evoluční algoritmy, Rekonfigurovatelné obvody, Vyvíjející se obvody, Číslicová filtrace, Obvodová realizace vyvíjejících se obvodů, Obvodová realizace číslicových filtrů, Evoluční návrh filtrů, Power PC, FPGA, Virtex II Pro

Poděkování

Na tomto místě bych rád poděkoval Ing. Lukáši Sekaninovi, Ph.D. za jeho odbornou pomoc a cenné rady při zpracování této práce. Dík patří i Ing. Tomáši Martínkovi za pomoc s praktickou realizací uvnitř FPGA.

Abstract

This project deals with a practical implementation of an evolvable hardware. The goal is to design and implement a complete hardware solution that utilizes a reconfigurable gate array FPGA Virtex II Pro as a target platform. By complete hardware implementation we mean that the evolutionary algorithm as well as the reconfigurable circuit are implemented in hardware. The primary advantage of this approach is high speed. The proposed system will be demonstrated on task of evolutionary design of image operators. Evolutionary algorithm is used to find the filter which minimizes the difference between the filtered image and desired training image. Because a complete hardware solution of this problem exists we make an effort to design a system that gives a better performance. We decided to use a PowerPC processor integrated in FPGA which allows us implement more efficient evolutionary algorithm in comparison to the hardware solution. The time-consuming evaluation part of evolutionary algorithm is running in the hardware and the remaining part in the software. The results indicate that the proposed solution outperforms the existing solution. Furthermore, general image operators for edge detection and impulse noise removal were discovered.

Keywords

Evolutionary algorithms, Reconfigurable hardware device, Evolvable hardware, Digital filters, Hardware implementation of evolvable hardware, Hardware implementation of digital filters, Evolutionary filter design, Power PC, FPGA, Virtex II Pro

Obsah

1 Úvod **7**

2 Evoluční algoritmy **8**
2.1 Úvod . . . 8
2.2 Genetické algoritmy . . . 9
2.3 Genetické programování . . . 10
2.4 Evoluční strategie . . . 10
2.5 Kartézské genetické programování . . . 11
2.5.1 Kódování obvodu . . . 11
2.5.2 Evoluční algoritmus . . . 12
2.5.3 Fitness funkce . . . 13

3 Rekonfigurovatelné obvody **14**
3.1 Návrh číslicových obvodů . . . 14
3.2 Architektura PLA . . . 15
3.3 Architektura CPLD . . . 15
3.4 Architektura FPGA . . . 16
3.4.1 XC6200 . . . 17
3.4.2 Virtex 4 . . . 17

4 Vyvíjející se obvody **18**
4.1 Evolvable Hardware . . . 18
4.2 Vlastnosti vyvíjejících se obvodů . . . 19
4.3 Klasifikace evolvable hardware . . . 20

5 Obvodová realizace vyvíjejících se obvodů **23**
5.1 Kritéria pro úspěšnou platformu . . . 23
5.2 Analogové obvody . . . 24
5.3 Číslicové obvody . . . 25
5.3.1 Specifické EHW obvody . . . 25
5.3.2 Využití FPGA . . . 26
5.3.3 VRC . . . 26

6 Číslicová filtrace **28**
6.1 Filtrace jednorozměrného signálu . . . 29
6.1.1 Lineární číslicové filtry . . . 29
6.1.2 Nelineární číslicové filtry . . . 31
6.2 Filtrace obrazu . . . 33
6.2.1 Lineární filtry . . . 33
6.2.2 Nelineární filtry . . . 35

7 Obvodová realizace číslicových filtrů 38
7.1 Řešení vycházející z konvoluce . . . 39
7.1.1 DSP procesory . . . 39
7.1.2 HW akcelerace . . . 40
7.2 Specifický přístup . . . 41
7.2.1 Filtry bez násobiček . . . 41
7.2.2 Filtrace na úrovni hradel . . . 42

8 Architektura PowerPC 44
8.1 PowerPC 405 . . . 45
8.1.1 Rozhraní řízení hodin CPM . . . 46
8.1.2 Rozhraní resetu RST a inicializace . . . 46
8.1.3 Sběrnice PLB . . . 47
8.1.4 Rozhraní OCM . . . 49
8.1.5 Rozhraní DCR . . . 50
8.1.6 Řadič přerušení EIC a obsluha přerušení . . . 51
8.1.7 Shrnutí . . . 52

9 Rekonfigurovatelný systém 53
9.1 Možnosti realizace . . . 53
9.2 Praktická realizace . . . 57
9.3 Blok PPC . . . 59
9.3.1 CLK . . . 61
9.3.2 IS PLB . . . 62
9.3.3 DS PLB . . . 63
9.3.4 DCR . . . 63
9.3.5 IS OCM . . . 64
9.3.6 DS OCM . . . 66
9.4 Blok PMI . . . 68
9.5 Blok VRC . . . 69
9.6 Blok FU . . . 70

10 Použitý evoluční algoritmus 73
10.1 Horolezecký algoritmus . . . 73
10.2 Populačně orientovaný algoritmus . . . 74
10.3 Paralelní algoritmus . . . 75

11 Evoluční návrh obrazových filtrů 76
11.1 Platforma . . . 76
11.2 Výsledek syntézy . . . 77
11.3 Experimentální vyhodnocení . . . 78
11.3.1 Evoluční algoritmus . . . 78
11.3.2 Generátor náhodných čísel . . . 81
11.3.3 Četnost mutace . . . 82
11.3.4 Generalizace . . . 82

12 Závěr 85

1 Úvod

Každoročně můžeme sledovat dramatické zvyšování výkonnosti elektronických obvodů, které již několik desetiletí odpovídá Moorovu zákonu. Obvody pracují stále na vyšším kmitočtu a díky novým technologiím se dosahuje vyššího stupně integrace. Na čip můžeme nyní umístit mnohonásobně více logiky, než tomu bylo před několika lety. Jako příklad vysoké integrace lze uvést běžné procesory, které v dnešní době obsahují přes miliardu tranzistorů na čipu. Návrh takovýchto obvodů se bohužel stává stále komplikovanějším a díky tomu může být do obvodu zaneseno během návrhu mnoho chyb. Navíc nikdo není schopen zaručit, že získaný obvod řeší daný problém optimálně. Se zvyšující se složitostí obvodu je návrhář nucen přecházet na vyšší úroveň abstrakce, se kterou je schopen efektivně pracovat. Tím však vznikají nové problémy, neboť zanedbáním některých detailů může vzniklý obvod získat nové, většinou negativní, vlastnosti. Obvod např. nemusí být navržen tak efektivně, jako kdyby byl navržen na úrovni tranzistorů. Díky špatnému návrhu může vznikat rušení, se kterým se nepočítalo apod. Ukazuje se, že oblast vyvíjejících se obvodů se jeví jako nadějné řešení tohoto problému v řadě aplikačních oblastí [18]. Z velmi abstraktní behaviorální specifikace obvodu je schopen evoluční algoritmus nalézt obvod, který vykazuje požadované chování.

Tato práce se zabývá hardwarovou realizací vyvíjejícího se obvodu (evoluční platformy) využívající procesor PowerPC. Jako cílová platforma bylo zvoleno rekonfigurovatelné hradlové pole VirtexII Pro, které kromě programovatelné logiky obsahuje i dva PowerPC procesory. Snahou bude navrhnout generickou evoluční platformu, pomocí které bude možno řešit řadu problému z oblasti evolučního návrhu. Činnost systému demonstrujeme prostřednictvím úlohy evolučního návrhu obrazových operátorů. Jelikož existuje čistě hardwarová realizace tohoto problému [20], pokusíme se systém navrhnout tak, abychom dosáhli minimálně stejného počtu řešení ohodnocených během jedné sekundy.

Struktura této práce je následující. Druhá kapitola se věnuje evolučním algoritmům, které tvoří základ vyvíjejících se obvodů. V této kapitole zmíníme evoluční algoritmy, které jsou v oblasti vyvíjejících se obvodů nejpoužívanější. Následující kapitola se stručně věnuje problematice rekonfigurovatelných obvodů, jsou uvedeny některé typy rekonfigurovatelných součástek a rozebrána jejich architektura. Čtvrtá kapitola se zabývá vyvíjejícími se obvody z teoretického hlediska a poskytuje přehled základních parametrů, podle kterých je možné tyto obvody klasifikovat. V páté kapitole jsou rozebrány některé z nejběžnějších realizací vyvíjejících se obvodů. Další kapitola poskytuje úvod do problematiky číslicové filtrace jednorozměrného a dvourozměrného signálu. Jsou uvedeny některé z běžně používaných filtrů a ukázány odezvy na základní typy šumu, které budou sloužit k porovnání s výsledky získanými pomocí evolučně návržených filtrů. Sedmá kapitola se věnuje běžně používaným i netradičním obvodovým realizacím číslicových filtrů. Osmá kapitola popisuje architekturu a sběrnice procesoru PowerPC použitého v hradlovém poli VirtexII Pro. Devátá kapitola popisuje hardwarovou implementaci navrženého rekonfigurovatelného systému. Desátá kapitola se zabývá možnostmi realizace evolučního algoritmu tak, aby byl získaný systém maximálně efektivní. Předposlední kapitola je věnována praktickému využití navržené platformy v aplikaci evolučního návrhu obrazových filtrů. Cílem je na základě experimentů porovnat navržený systém s dostupnou hardwarovou platformou, která využívá jednoduchý evoluční algoritmus.

2 Evoluční algoritmy

Jelikož je problematika evolučních algoritmů velmi rozsáhlá, věnuje se tato kapitola zejména těm algoritmům, které jsou v oblasti vyvíjejících se obvodů nejčastěji používané. Detailněji se však zaměříme pouze na kartézské genetické programování, které bude použito v navrhovaném vyvíjejícím se obvodu. Ucelený přehled všech známých algoritmů můžeme nalézt např. v [9, 18].

2.1 Úvod

Evoluční algoritmus je stochastický prohledávací algoritmus inspirovaný Darwinovou teorií evoluce. Prohledávaný prostor je množina všech uvažovaných řešení určitého problému, prvek této množiny představuje jedno řešení. Rozdíl oproti klasickým prohledávacím algoritmům jako je např. náhodné prohledávání nebo horolezecký algoritmus [9] spočívá v tom, že evoluční algoritmy obvykle pracují nad množinou prvků, kterou nazýváme populace kandidátních řešení. Nová populace vznikne aplikací speciálních operátorů na původní populaci a působením tzv. selekčního tlaku, který směřuje evoluci směrem k lepším řešením v prohledávaném prostoru. Nejčastěji používanými operátory jsou křížení a mutace, ty jsou inspirovány procesy probíhajícími v přírodě. Informaci o úspěšnosti jednotlivých řešení získá evoluční algoritmus díky fitness funkci. Fitness hodnota řešení vypočítaná pomocí fitness funkce udává, jak splňuje toto řešení požadavky. Základní struktura evolučního algoritmu je následující:

```
čas t = 0;
vytvoření počáteční populace P(t);
vyhodnocení fitness všech prvků P(t);
while (není splněna podmínka ukončení) {
  t = t + 1;
  P(t) = vytvoření nové populace z předchozí populace P(t-1);
  vyhodnocení fitness všech prvků P(t);
}
```

Základními představiteli evolučních algoritmů jsou evoluční strategie, evoluční programování, genetické algoritmy a genetické programování. Všechny techniky byly vyvinuty nezávisle na sobě a liší se především ve způsobu kódování řešení do tzv. chromozomu a realizací evolučních operátorů.

Oblast aplikace evolučních algoritmů je velmi rozsáhlá – od nalezení optima n-rozměrné funkce až po návrh analogových nebo digitálních obvodů. Z tohoto důvodu je vhodné rozlišovat mezi prostorem fenotypu a genotypu. Libovolné řešení (genotyp) je po dekódování mapováno na odpovídající řešení (fenotyp). Na fenotyp (nikoliv na genotyp) je následně aplikována fitness funkce. Abychom si tyto termíny přiblížili, můžeme se podívat na příklad kódování obvodu v chromozomu (genotyp) a odpovídající schéma obvodu (fenotyp) v kapitole 2.5.1. Reprezentační prostor obsahuje řetězce (vektory), které kódují prvky vyhledávacího prostoru tak, aby aplikace genetických operátorů byla co nejsnazší. Zobrazení genotyp-fenotyp se často označuje jako růstová funkci (growth function). Řetězce reprezentující prvky prostoru genotypu se označují jako chromozomy (jedinci). Každý chromozom sestává z určitého počtu genů, hodnota genu se nazývá alela. Počet genů v chromozomu závisí na charakteru řešeného problému, dokonce může být i proměnný.

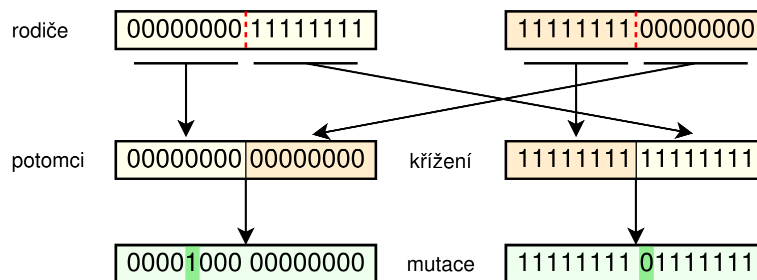
2.2 Genetické algoritmy

Genetické algoritmy patří mezi nejznámější a nejpoužívanější evoluční algoritmy [9]. Nejjednodušší forma genetického algoritmu, tzv. kanonický genetický algoritmus, je na následujícím výpisu.

```
čas t = 0;
náhodná inicializace chromozomů v P(t);
do {
  vyhodnocení fitness všech jedinců v P(t);
  vyber jedince do prostoru páření;
  t = t + 1;
  do {
    náhodně vyber dva rodiče z prostoru páření;
    pomocí operátoru krížení vytvoř dva potomky;
    aplikuj náhodnou mutaci na potomky;
    utvoř P(t) nahrazením jedinců v P(t-1);
  } until (P(t) je naplněna novými potomky);
} until (splněna podmínka ukončení);
```

Tento algoritmus tradičně pracuje s vektorem (chromozomem) pevné délky skládajícím se z genů, které mohou mít formu binárních hodnot, celočíselných hodnot, znaků nebo reálných hodnot.

Nejjednodušším operátorem křížení je jednobodové křížení, které je uvedeno na obrázku 1. Nejprve se náhodně určí bod křížení, který rozdělí chromozom na dvě části. V našem případě je bod křížení roven polovině délky chromozomu. V dalším kroku se provede záměna levé a pravé části obou rodičů, čímž vzniknou ze dvou rodičů dva potomci, jejichž chromozom se skládá z obou rodičů. Jako většina dějů řídí se i křížení pravděpodobností. Obyčejně se v 70% případů křížení použije, ve zbylých 30% případů jsou potomci identicky rovni rodičům, což odpovídá klonování. Mutace se podobně jako v přírodě vyskytuje velmi zřídka a provádí změnu hodnoty genu na opačnou.



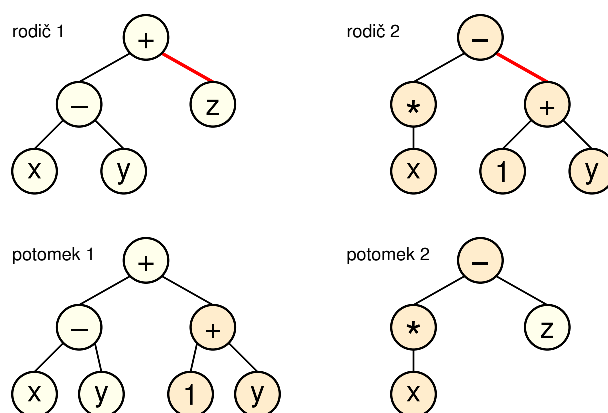
Obrázek 1: Operátory jednobodového křížení a mutace používané v genetických algoritmech

Výběr potomků (operátor selekce) je většinou implementován jako pravděpodobnostní operátor využívající normalizovanou fitness hodnotu jedince x_i jako pravděpodobnost jeho výběru $p(x_i)$. Tato metoda je známa jako ruletový výběr (rulette wheel) [9].

2.3 Genetické programování

Genetické programování může být chápáno jako forma genetického algoritmu, který operuje nad chromozomy proměnné délky s využitím modifikovaných operátorů [9]. Genetické programování bylo vyvinuto zejména pro automatický návrh programů a symbolickou regresi. Narozdíl od většiny evolučních algoritmů se nerozlišuje mezi genotypem a fenotypem.

Hledaný program je reprezentován buď stromem nebo lineární formou pomocí strojových instrukcí. Operátor křížení je realizován jako výměna podstromů. Náhodně se v obou rodičovských stromech stanoví body křížení a poté se vymění podstromy, které body křížení vymezují. Celý proces je znázorněn na obrázku 2. Operátor mutace zvolí náhodně podstrom a nahradí jej jiným náhodně vygenerovaným podstromem.



Obrázek 2: Operátor křížení používaný v genetickém programování

2.4 Evoluční strategie

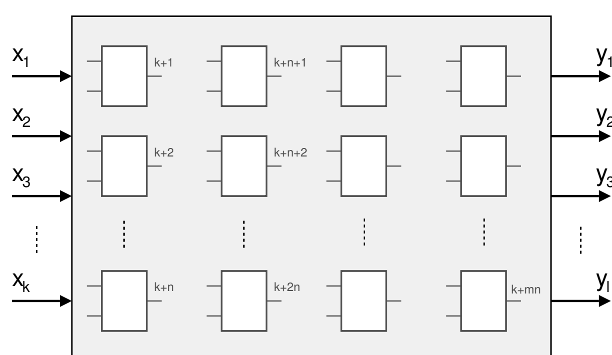
Evoluční strategie (ES) byla vyvinuta především pro řešení optimalizačních úloh. Podobně jako většina genetických algoritmů rozlišuje evoluční strategie mezi genotypem a fetonypem. Každý jedinec je reprezentován vektorem reálných hodnot. Důležitou roli hraje operátor mutace, který je považován za primární operátor. Operátor mutace přičte k některému genu náhodnou hodnotu, která se řídí normálním rozložením se středem v nule a předem zvolenou standartní odchylkou. Hlavní vlastností, která však odlišuje evoluční strategie od ostatních algoritmů, je obsah chromozomu. Chromozom obsahuje kromě standartní konfigurace i řídicí parametry jako je standartní odchylka a úhel natočení, které se vyvíjí současně s vývojem jedince. Tento přístup lze označit jako samoadaptace.

Rozlišujeme dva druhy selekce označované jako $(\mu + \lambda)$-ES a (μ,λ)-ES. Oba selekční mechanismy jsou deterministické. Strategie $(\mu + \lambda)$-ES vybere μ nejlepších jedinců z populace rodičů i potomků. Tito jedinci budou tvořit novou rodičovskou populaci. Strategie (μ,λ)-ES vybírá μ nejlepších jedinců pouze z populace potomků.

2.5 Kartézské genetické programování

Kartézské genetické programování (CGP) bylo představeno poprvé v roce 1999 v článcích [15, 11]. Číslicový obvod je reprezentován maticí $m \times n$ programovatelných elementů (hradel), která má k vstupů a l výstupů (viz obrázek 3). Cílem evolučního algoritmu je nalézt konfiguraci těchto elementů a jejich vzájemné propojení.

V našem případě budeme uvažovat programovatelné elementy, které mají dva vstupy a jeden výstup. Nad touto maticí se hledá vhodné propojení jednotlivých elementů tak, aby výstup vzniklého kombinačního obvodu byl pro všechny vstupní kombinace totožný se zadanou pravdivostní tabulkou a kombinační obvod tak plnil požadovanou funkci.



Obrázek 3: Rekonfigurovatelný obvod

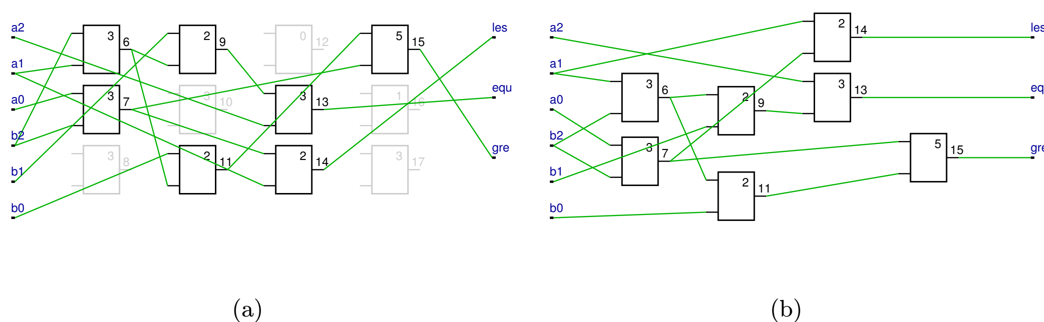
Programovatelné elementy mohou realizovat jednu z r funkcí. V případě kombinačních obvodů to jsou booleovské funkce AND, OR, XOR, NOT, NAND, NOR, atd. Aby nemohla vzniknout zpětná vazba, jejíž vyhodnocení by se stalo velmi obtížným, mohou se propojit vstupy elementu pouze na výstupy elementů v předchozích sloupcích nebo na kterýkoliv vstup kombinačního obvodu. Parametr řídící vnitřní konektivitu se nazývá $l - back$ parametr. Parametr určuje, o kolik sloupců vpřed je možno připojit vstup elementu. Jeho rozsah hodnot se může pohybovat v intervalu $[1, m]$. Hodnota 1 znamená, že je možné vstup elementu zapojit pouze na některý z výstupů předchozího sloupce, naopak maximální hodnota m říká, že je možné vstup elementu připojit na jakýkoliv výstup předchozích sloupců. Maximální hodnota $l - back$ parametru zajistí maximální konektivitu a tím možnost hledání velmi komplexních obvodů, naproti tomu zvolíme-li hodnotu 1, je možné výsledný obvod velmi lehce převést do zřetězeného provozu (pipeline). Výstupy kombinačního obvodu je možné připojit na výstup kteréhokoliv elementu.

2.5.1 Kódování obvodu

Chromozom tvoří lineární řetězec celočíselných hodnot délky $mn(i + 1) + l$, kde i je počet vstupů elementu a l je počet výstupů kombinačního obvodu. Každý výstup elementu a každý z k vstupů kombinačního obvodu mají přiřazeno své číslo. Číslování začíná od vstupů kombinačního obvodu, poté následují jednotlivé programovatelné elementy (po sloupcích zleva doprava).

Chromozom je složen z $m \times n$ trojic celočíselných hodnot, kde první dvě hodnoty udávají číslo výstupu, na který je připojen první a druhý vstup trojici příslušejícího elementu, třetí hodnota udává jakou logickou funkci element realizuje. Na konci chromozomu je $l - tice$ o velikosti shodné s počtem výstupů kombinačního obvodu. Každý prvek l-tice určuje index elementu (jeho výstupu), na který bude primární výstup obvodu připojen.

Budeme uvažovat následující tvar chromozomu: (3,1,3) (2,3,3) (2,0,3) (4,6,2) (4,4,3) (5,6,2) (9,8,0) (9,0,3) (7,1,2) (11,7,5) (7,11,1) (11,12,3) (14,13,15). Tento chromozom kóduje dvouvstupový rozšířený komparátor v rekonfigurovatelném poli 4×3 elementů. Chromozomu odpovídá propojení elementů matice a přiřazení funkcí, které je zobrazeno na obrázku 4a. Na obrázku 4b je překreslená varianta s vynechanými nepoužitými elementy, které se nepodílejí na řešení.



(a) (b)

Obrázek 4: 2-bitový komparátor a) rozmístění v matici b) překreslené schema. Přiřazené funkce: 1-AND, 2-OR, 3-XOR.

2.5.2 Evoluční algoritmus

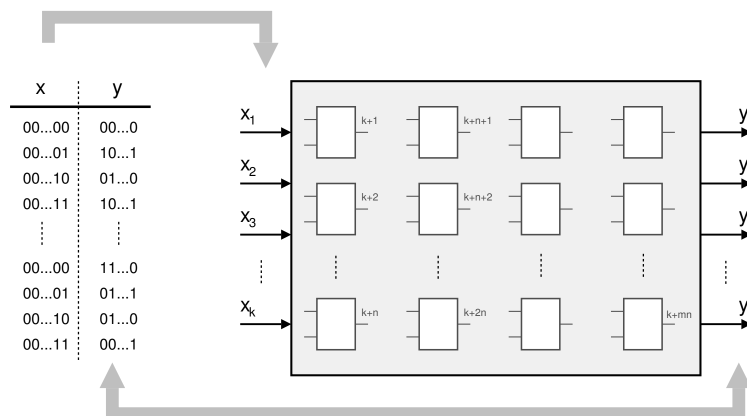
Algoritmus kartézského genetické programování je založen na evoluční strategii (1+λ)-ES [9]. Pro CGP se stejně jako u ES využívá pouze operátor mutace, který však pracuje na jiném principu. Tento operátor je řízen parametrem udávajícím procentuální počet mutovaných genů (jeden gen odpovídá jedné celočíselné hodnotě chromozomu). Operátor náhodně změní hodnotu některého genu na jinou.

Algoritmus lze popsat následujícími kroky:

1. Vygenerování 1+λ náhodných jedinců (chromozomů) pro inicializaci populace,
2. Ohodnocení všech jedinců populace pomocí fitness funkce,
3. Nalezení nejlépe ohodnoceného jedince (nejvyšší fitness),
4. Vygenerování λ potomků mutací nalezeného nejlepšího jedince,
5. Nalezený nejlepší jedinec společně s jeho λ potomky tvoří novou populaci,
6. Není-li splněna podmínka ukončení, pokračuje se krokem 2.

2.5.3 Fitness funkce

Protože během evoluce je nutné posuzovat funkčnost zapojení, je zapotřebí funkce, která stanoví, jak dobré je konkrétní zapojení. Taková funkce se nazývá fitness funkce a je mírou funkčnosti zapojení. Výpočet fitness hodnoty pro konkrétní chromozom se provede tak, že se vyhodnotí funkčnost zapojení postupným nastavováním všech možných vstupních kombinací daných pravdivostní tabulkou, jak ukazuje obrázek 5.



Obrázek 5: Vyhodnocení fitness

V případě úplné definice se jedná o 2^k kombinací, kde k je počet vstupů kombinačního obvodu. Pro nastavenou vstupní kombinaci se vypočítají v závislosti na propojení elementů hodnoty výstupů. Simuluje se funkce předloženého kombinačního obvodu a porovnává se výstupní kombinace z dané pravdivostní tabulky s odsimulovanou výstupní kombinací. Hodnota fitness pak odpovídá počtu shod s předem zadanými výstupními hodnotami pro všechny vstupní kombinace. Pokud máme např. $k = 4$ vstupy a $l = 2$ výstupy, pak počet vstupních kombinací je $2^k = 2^4 = 16$. Maximální fitness pak může být $l \cdot 2^k = 2 \cdot 16 = 32$ shod. V praxi lze simulaci urychlit tím, že se počítá nikoliv v bitech, ale v 32 bitové aritmetice, čímž se vyhodnocování $32\times$ urychlí a tím sníží řád složitosti o 2^5. Kromě této jednoduché fitness funkce, která posuzuje kvalitu zapojení pouze podle funkčnosti, existuje i objektivní fitness, která zohledňuje i jiné důležité aspekty [27]. Jedná se např. o počet použitých elementů, spotřebu, zpoždění, atd.

3 Rekonfigurovatelné obvody

Cílem této kapitoly je popsat princip rekonfigurovatelných obvodů. Kapitola je zaměřena na několik rekonfigurovatelných obvodů, zejména programovatelných hradlových polí FPGA (Field Programmable Gate Array), produkovaných firmou Xilinx [41]. Na trhu sice existuje několik dalších producentů zabývajících se výrobou rekonfigurovatelných obvodů (např. Atmel a Altera), ale firma Xilinx je v této oblasti průkopníkem a zaujímá největší podíl na trhu.

Rekonfigurovatelný obvod je hardware, který je možno pomocí konfiguračního řetězce nastavit tak, aby realizoval požadovanou funkci. Nejjednodušší rekonfigurovatelný obvod si můžeme představit jako sadu hradel, které mohou realizovat jednu ze dvou funkcí – např. funkci OR nebo funkci AND. Určení funkce, kterou bude hradlo realizovat, závisí na hodnotě bitu v konfiguračním řetězci, která hradlu přísluší. V konfiguračním řetězci kromě nastavení jednotlivých hradel musí být i informace o jejich vzájemném propojení. Konfigurační informace určuje, jak se naprogramují jednotlivé hradla rekonfigurovatelného obvodu a kam se mají připojit jejich vstupy a výstupy. Pomocí tohoto řetězce, který je většinou uložen v paměti RAM případně EEPROM, máme možnost vytvořit uvnitř dnes dostupných rekonfigurovatelných obvodů libovolný číslicový obvod. V případě, že potřebujeme jiný obvod, stačí pouze nahrát jiný konfigurační řetězec.

3.1 Návrh číslicových obvodů

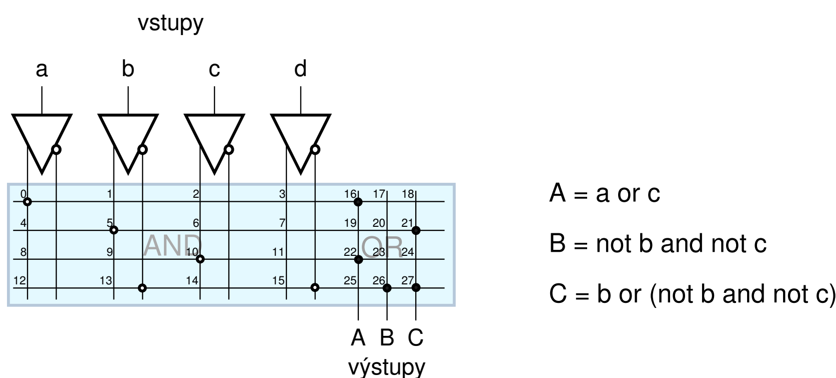
Motivací pro zavedení rekonfigurovatelných obvodů byla snaha o zjednodušení návrhu složitých obvodů a především snaha o zkrácení doby návrhu, neboť ta určuje výslednou cenu a okamžik uvedení na trh. Snahou je transformovat proces návrhu číslicových obvodů tak, aby se co nejvíce přiblížil návrhu software. Trendem je přejít postupně k návrhu obvodů pomocí jazyků s vysokou úrovní abstrakce. Příkladem může být jazyk HandelC, který umožňuje pomocí standartních konstrukcí jazyka C a několika specifických výrazů popsat libovolný hardware. Vytvořit obvod ovšem není úplně triviální a přímočaré, je zapotřebí stále myslet na to, že procesy v hardware narozdíl od instrukcí programu probíhají většinou paralelně.

Moderní proces návrhu obvodů se skládá z následujících kroků. Nejprve se pomocí některého z jazyků pro popis hardware vytvoří specifikace obvodu. Mezi nejznámější jazyky patří VHDL, Verilog a HandelC. Podle této specifikace je možné provést simulaci na funkční úrovni. Dalším krokem je syntéza, jejímž úkolem je detekovat známé programové konstrukce vedoucí např. na registr, sčítačku, apod. a vytvořit schema složené ze stavebních bloků FPGA a odpovídající konfigurační řetězec. Během syntézy můžeme získat simulační model obvodu, který koresponduje s chováním FPGA naprogramovaného tímto řetězcem. S využitím získaného simulačního modelu je možné provést přesnou tzv. časovou simulaci. Po syntéze stačí pouze nahrát konfigurační řetězec do rekonfigurovatelného obvodu a získáme navrhovaný obvod.

Díky jazyku s vyšší úrovní abstrakce je možné ve velmi krátké době vytvořit prototyp zařízení v FPGA, který může být po fázi testování nasazen do reálného provozu. Získané schema může také sloužit jako předloha pro výrobu ASIC čipu.

3.2 Architektura PLA

Mezi první programovatelné obvody patří obvody typu PLA (Programmable Logic Array). Tyto obvody se skládají ze dvou programovatelných rovin – roviny AND a roviny OR. Struktura obvodu je znázorněna na obrázku 6. V AND rovině máme možnost vytvářet libovolný logický součin složený ze vstupních signálů a jejich negací. V rovině OR se vytváří logický součet, jehož vstupy jsou složeny z logických součinů získaných v rovině AND. Tato rovina obsahuje tolik signálů, kolik je výstupních vodičů.



Obrázek 6: Struktura PLA obvodu. V rovině AND se vytváří na vodorovných vodičích logický součin, v rovině OR se na svislých vodičích odpovídajících jednotlivým výstupům vytváří logický součet

Pomocí PLA máme možnost realizovat libovolnou kombinační logickou funkci. Jediným omezením je počet vodičů jednotlivých rovin.

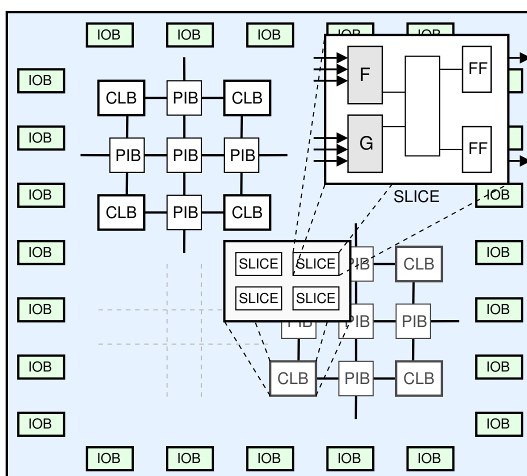
3.3 Architektura CPLD

PLA a následně PAL obsahující klopné obvody se staly inspirací pro vznik mnohem složitějších a sofistikovanějších obvodů CPLD (Complex Programmable Logic Device), které jsou nyní současně s FPGA nejpoužívanějšími programovatelnými obvody. Konfigurační řetězec je uložen většinou v paměti EEPROM, která je součástí obvodu. Uvnitř CPLD můžeme nalézt hierarchickou strukturu, avšak mnohem jednodušší než se nachází v obvodech FPGA.

Základní stavební jednotkou je makrobuňka (macrocell), která umožňuje vytvářet několik logických součtů. Kromě toho každá makrobuňka obsahuje jeden klopný obvod typu RS, logiku pro připojení ke globálnímu nulování a globálnímu hodinovému signálu a jeden ze signálů je napojen na I/O pin. Větším celkem je makroblok, který sdružuje několik makrobuněk. Na úrovni makrobloku se vytvářejí logické součiny (podobně jako u PLA). Běžně dostupné CPLD obvody mohou pracovat na kmitočtu 200MHz a obsahují kolem 300 makrobuněk, což představuje asi 7000 využitelných logických hradel.

3.4 Architektura FPGA

Jak jsme se již zmínili v předchozím odstavci, představují současné FPGA obvody špičku na poli programovatelných obvodů. FPGA obvod typicky tvoří dvourozměrné pole programovatelných elemetů CLB (Configurable Logic Block) a PIB (Programmable Interconnect Block), viz obrázek 7. Po obvodu FPGA jsou umístěny programovatelné IOB (Input/Output Block) elementy, které jsou napojeny na jednotlivé piny. PIB elementy umožňují jednak propojení s IOB bloky a jednak propojení jednotlivých CLB elementů navzájem. CLB blok se skládá z



Obrázek 7: Struktura FPGA obvodu rodiny Virtex

několika menších logických elementů označovaných jako SLICE. Jádro základní stavební jednotky SLICE tvoří dva funkční generátory F a G, které jsou doplněny několika multiplexory a dvěma registry. Funkční generátor je implementován jako programovatelná lookup tabulka LUT a může realizovat libovolnou logickou funkci, jenž má čtyři vstupy a jeden výstup. Pomocí multiplexorů lze v rámci jednoho CLB vytvářet složitější logické funkce. SLICE obsahuje i systém pro konstrukci rychlých sčítaček, čítačů, komparátorů apod. Bloky PIB je možné také programovat, přinejmenším je zapotřebí stanovit orientaci pinu – vstupní, výstupní nebo vstupně-výstupní a logické úrovně, se kterými bude pin pracovat.

Kromě těchto základních elementů jsou v FPGA obvodu rozmístěny i složitější celky – např. 18kb dvouportové blokové paměti, 18 bitové násobičky a dokonce i několik PowerPC procesorů.

V dnešní době existují na trhu dvě rodiny FPGA obvodů Xilinx – rodina Spartan a výkonější rodina Virtex. Struktura rekonfigurovatelného pole obou rodin FPGA je stejná a odpovídá popisu výše. Hlavní rozdíl je v přítomnosti a počtu rozšiřujících obvodů jako jsou blokové paměti, násobičky, gigabitové transceivery apod. Cílem je nabídnout obvody, které mají velmi dobrý poměr ceny a výkonu (Spartan 3) a obvody, které jsou sice dražší, avšak mohou poskytnout velmi vysoký výkon (Virtex, Virtex II Pro, Virtex 4).

Jelikož se z pohledu vyvíjejících se obvodů jedná o velmi zajímavou architeturu, zmíníme se v následujících odstavcích o jedné ze starších rodin FPGA – rodině XC6200. Kapitolu uzavřeme stručnou zmínkou o nejnovější rodině FPGA – Virtex 4.

3.4.1 XC6200

Rodina FPGA obvodů XC6200 je jednou z unikátních rodin. Obvod podporuje částečnou rekonfiguraci, celý formát konfiguračního řetězce je zdokumentován a je k dispozici uživateli. Další pozitivní vlastností je, že náhodně generovaný konfigurační řetězec nemůže obvod zničit. Tyto FPGA obvody splňují všechny předpoklady pro úspěšné využití jako platformy pro vyvíjející se obvod. Jediným nedostatkem je, že tato rodina již není na trhu k dispozici, neboť byla vytlačena mnohem výkonnější a komerčně úspěšnější rodinou Virtex.

3.4.2 Virtex 4

Rodina Virtex 4 je zástupcem nejmodernějších rekonfigurovatelných obvodů. Existují celkem tři druhy obvodů podle cílového použití:

- Virtex 4 LX - vytvořen pro aplikace vyžadující velké množství logiky, obvod obsahuje pouze logiku a paměti,
- Virtex 4 FX - vytvořen pro komplexní vestavěné systémy, oproti předchozí verzi obsahuje FPGA navíc PowerPC procesory a RocketIO transceivery,
- Virtex 4 SX - vytvořen pro DSP aplikace, FPGA obsahuje rychlé násobičky a velké množství vestavěných multiply-and-accumulate bloků, které jsou základní výpočetní jednotkou při zpracování signálů.

Obvody jsou vystavěny na moderní 90 nm technologii. Logika i paměti uvnitř FPGA mohou pracovat až do 500 MHz. Struktura základního bloku CLB zůstala zachována a odpovídá popisu výše. Procesory PowerPC mají vyvedenou sběrnici APU (Auxiliary Processor Unit), pomocí které je možné se připojit na dekódovací a exekuční jednotku procesoru. Uživatel má možnost vytvořit své vlastní instrukce, které budou akcelerovány pomocí logiky v FPGA.

Všechny současně dostupné obvody rodiny Virtex je možné programovat několika způsoby. Prvním nejčastěji používaným způsobem je konfigurace pomocí externího zařízení, které je připojeno na k tomuto účelu sloužící konfigurační port. Standartně se používá přeprogramování celého FPGA obvodu, což znamená nahrát kompletní konfigurační řetězec. Kromě tohoto postupu je však možné využít hardwarové podpory pro částečnou rekonfiguraci, která umožňuje nahrát pouze část konfiguračního řetězce. Dalším způsobem je interní rekonfigurace, která je řízena zevnitř FPGA. K tomu je určen konfigurační port ICAP (Internal Configuration Access Port), který je možné napojit na vlastní obvod uvnitř FPGA.

4 Vyvíjející se obvody

Základní myšlenkou evolvable hardware je spojení rekonfigurovatelné součástky s evolučními algoritmy. Cílem je získat obvod, který bude mít možnost se během své činnosti sám rekonfigurovat a přizpůsobovat se např. změně prostředí. Jinou možností je pomocí evoluce a rekonfigurovatelné součástky vyvinout obvod, který bude splňovat požadavky, které jsme do systému vložili, tzn. získaný obvod bude pracovat ve statickém prostředí. Jedná se o poměrně nový přístup, jehož inspirací byla evoluce probíhající v přírodě. Evolvable hardware lze charakterizovat jako technologii umožňující vytvořit vyvíjející se systém, který je chopen autonomně a dynamicky měnit svou vnitřní konfiguraci v závislosti na změně prostředí, v němž je použit.

4.1 Evolvable Hardware

V posledních letech se v oblasti evolvable hardware (EHW) striktně rozlišuje mezi evolučním návrhem obvodů a vyvíjejícím se obvodem [4]. V případě evolučního návrhu obvodů [3] je výsledkem evoluce jeden obvod. Evoluční algoritmus je použit pouze během fáze návrhu, zastává tedy roli návrháře obvodu. Tento přístup je pro nás zajímavý z toho hlediska, že do evolučního algoritmu můžeme zabudovat libovolné kritérium – např. minimalizaci ceny výsledného obvodu případně maximalizaci kvality obvodu. V případě evolučního návrhu obvodů je zpravidla zapotřebí mnoha evolučních kroků, doba evoluce pro nás proto nesmí být kritická. Pokud potřebujeme navrhnout velmi specifický obvod, který bychom klasickými konvenčními technikami nebyli schopni vytvořit, vyplatí se určitou dobu počkat. Získaný obvod může být poté několik let využíván.

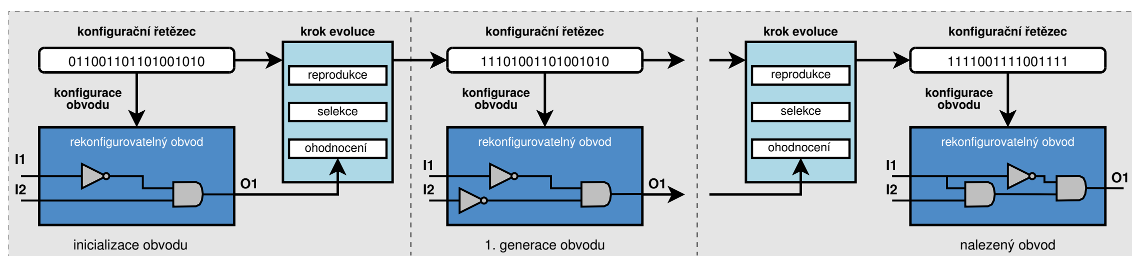
V případě, že bude celý systém implementován v hardware, jedná se o tzv. vyvíjející se obvod. Tyto obvody většinou pracují v dynamicky se měnícím prostředí. Vyvíjející se obvody mohou zajišťovat buď adaptaci (např. v robotice) nebo realizovat vysoce výkonné výpočty (např. komprese obrazu), které jsou pomocí konvenčních postupů nedostupné. Vyvíjející se obvody je možné použít i pro zajištění opravy při poruše části obvodu. V kontrastu s evolučním návrhem obvodů je evoluční algoritmus součástí systému. Úlohou není nalézt nový obvod, ale poskytovat dlouhodobě co nejvyšší výkon (kvalitu). Oproti evolučnímu návrhu není kritické ani množství použitých hradel, které obvod realizují.

Jiným přístupem je klasifikace podle délky životního cyklu. Pokud životní cyklus v určitý okamžik končí, pak se jedná o evoluční návrh. Na druhou stranu pokud cyklus nikdy neskončí, jedná se o vyvíjející se obvod. Z tohoto pohledu tedy můžeme evolvable hardware chápat jako obvod, který má možnost se sám rekonfigurovat, nebo jako metodu návrhu obvodů.

Klasické metody návrhu digitálních obvodů jsou většinou založeny na dekompozici řešeného problému a minimalizaci. Jedná se o tzv. přístup shora dolů, při němž má konstruktér možnost pracovat na několika úrovních abstrakce. U složitých návrhů se pracuje např. na úrovni meziregistrových přenosů, neboť návrh přímo na úrovni tranzistorů, případně hradel by byl velmi obtížný a časově náročný. Myšlenka evolvable hardware je zcela odlišná a pro konstruktéra velmi pohodlná. Místo manuálního návrhu se pouze specifikuje, jak se má požadovaný obvod chovat, nikoliv jakým způsobem se má realizovat. Specifikace může být např. vztahem mezi vstupem a výstupem, tzn. jak má obvod reagovat na určitou kombinaci na vstupu. Kromě specifikace můžeme zvolit stavební bloky obvodu, zadat omezující kritéria atd.

Obvod je poté automaticky navržen ve většině případů pomocí genetického algoritmu [9].

Algoritmus vyvíjejícího se obvodu je znázorněn na obrázku 8. V prvním kroku je vygenerována počáteční množina (populace) obvodů (jedinců), se kterou se bude během celé doby evoluce pracovat, neboť jako většina evolučních algoritmů je i tento algoritmus orientován populačně. Konfigurace jednotlivých obvodů počáteční množiny je generována ve většině aplikací náhodně. V dalším kroku je vyhodnoceno chování všech jedinců populace a každému je přiřazena tzv. fitness hodnota. Čím lépe obvod splňuje na něj kladené požadavky, tím je tato hodnota vyšší a naopak. V posledním kroku se vybere nejlepší kandidát, tedy jedinec s nejvyšší fitness hodnotou, ze kterého se použitím genetických operátorů (mutace, křížení) vytvoří nová populace jedinců. Podrobnosti týkající se operátorů jsou diskutovány v kapitole pojednávající o evolučních algoritmech. Nutno poznamenat, že je nutné si pamatovat dosud nejlepší nalezené řešení, jak ukazuje obrázek. Předchozí dva kroky se opakují stále dokola. Evoluci můžeme ukončit tehdy, splňuje-li nalezený obvod naše požadavky, případně byl-li dosažen limit počtu kroků. Pokud se jedná o vyvíjející se obvod pracující v dynamicky měnícím se prostředí, evoluční cyklus nikdy nekončí a stále se hledá lepší obvod.



Obrázek 8: Schema algoritmu vyvíjejícího se obvodu

Vlastní evoluce může trvat jen několik kroků. Potřebujeme-li v nějaké aplikaci obvod přizpůsobit novému prostředí, dovolíme evoluci např. sto kroků na nalezení nejlépe se chovajícího obvodu a s tímto obvodem poté pracujeme. Nic ovšem nebrání tomu, aby na pozadí docházelo k hledání lepšího řešení.

4.2 Vlastnosti vyvíjejících se obvodů

Obecně se ukazuje, že evoluce nám umožňuje navrhovat obvody, které jsou pro konvenční návrh nedosažitelné [29]. Evoluce se dokáže oprostit od tradičního postupu, který vnáší do návrhu řadu omezení a odstraňuje předsudky, které mohl vnést sám návrhář. Díky realizaci v hardware jsme schopni řešení nalézt rychleji, než konvenční metoda, případně poskytnout alespoň suboptimální řešení tam, kde klasická metoda z časových důvodů neposkytne žádné. Díky použití evoluce získáme systém, který má vrozenou odolnost proti poruchám. Pokud nastane porucha některého z elementů rekonfigurovatelné součástky, je velmi pravděpodobné, že se díky redundanci programovatelných elementů podaří sestavit obvod ze zbylých elementů. Může se také stát, že evoluce využije pro realizaci obvodu i chybně se chovající elementy. Tato vlastnost je z určitého pohledu velmi zajímavá a není možné ji simulovat (výhoda instrinsické evoluce – viz následující podkapitola).

Nebyly bychom objektivní, kdybychom se nezmínili i o nevýhodách. Největší překážkou je problém škálovatelnosti. V případě velkého počtu elementů konfigurovatelného obvodu je výpočet časově velmi náročný. Evoluce zatím zaznamela úspěch jen pro poměrně jednoduchá zapojení. Jako velmi slibná platforma se však jeví moderní rekonfigurovatelné architektury FPGA. V dnešní době tyto obvody nabízejí velmi velkou kapacitu (co se týče počtu elementů) i výkonnost (kmitočet řádově stovky MHz). Díky speciálnímu vysokorychlostnímu rozhraní lze jednotlivé FPGA obvody propojovat, čímž získáme velmi výkonný systém. Tento princip byl využit při realizaci vysokokapacitní neuronové sítě CAM-brain [35], kde bylo pomocí 72 FPGA dosaženo výpočetního výkonu srovnatelného s 10 000 procesory Pentium III 500 MHz. Nutno poznamenat, že použité FPGA obvody dosahují jen zlomku výkonnosti dnešních FPGA obvodů. Dalším nedostatkem evoluce je schopnost generalizovat. Při vyhodnocování funkčnosti logických obvodů se fitness funkce většinou počítá jako počet správně určených výstupů přes všechny požadované vstupní kombinace. Přidáním jednoho bitu na vstupu se počet vstupních vektorů zdvojnásobí, což má za následek zdvojnásobení doby potřebné k ohodnocení. Pro větší počet vstupů jsme nuceni omezit počet vstupních vektorů jen na určitou podmnožinu všech kombinací. Ukazuje se, že evoluční technika sice navrhne obvod, který správně reaguje na trénované vstupní kombinace, avšak chybně na zbylé kombinace. Posledním problémem je odolnost. Pokud jsme pomocí evoluce získali obvod, který je velmi složitý, musíme ho bez důkladného studia chápat pouze jako „černou skříňku“. Získaný obvod sice vykazuje požadované chování, ale o jeho dalších vlastnostech nevíme téměř nic, neboť obvod nebyl navržen klasickou metodou, ale metodou, která se řídí nám neznámými pravidly. V případě, že evoluce využije vlastností materiálu, bude obvod pevně spjat s prostředím, ve kterém byl vytvořen a v jiném prostředí bude nefunkční.

4.3 Klasifikace evolvable hardware

Výzkum v oblasti evolvable hardware je velmi různorodý. Abychom měli alespoň základní povědomí o jednotlivých oblastech výzkumu, poskytuje tato kapitola přehled základních parametrů, které máme možnost jakožto návrháři ovlivnit.

Evoluční algoritmus. Na místě evolučního algoritmu máme možnost volit ze široké škály, kterou tato oblast nabízí. Ve většině případů se však používá genetický algoritmus. Dalšími často používanými algoritmy jsou genetické programování a evoluční programování. O některých přístupech se podrobněji zmiňuje kapitola 2. Hlavní rozdíl spočívá v rozdílné reprezentaci obvodu. Genetické algoritmy pracují s lineárním konfiguračním řetězcem, genetické programování operuje nad stromem. Obě metody používají operátory křížení i mutace, evoluční programování využívá pouze operátor mutace.

Technologie rekonfigurovatelného obvodu. Rekonfigurovatelný obvod může být plně *analogový*, plně *digitální* případně *analogově-digitální*. U analogového obvodu se pracuje s klasickými analogovými součástkami – většinou se však jedná o ladění parametrů např. konfigurace tranzistorů, proudových zdrojů apod. Návrh analogového obvodu na úrovni základních analogových součástek je oproti digitálnímu návrhu komplikovanější, důvodem je náročnější simulace obvodu.

Architektura. Cílem evoluce může být *kompletní návrh obvodu* ze základních stavebních bloků. Jedná se o klasickou evoluci - hledání konfigurace jednotlivých elementů a jejich vzájemného propojení. Jednodušší aplikací je pouhé *dolaďování parametrů* (*parameter tunning*). V tomto případě konfigurační řetězec reprezentuje lineární pole obsahující jednotlivé parametry.

Stavební bloky. Evoluční návrh obvodů je možné realizovat na různé úrovni abstrakce. Nejnižší úrovní abstrakce je *evoluce na úrovni základních elementů* - u digitálních obvodů se jedná o hradla, u analogových o tranzistory a ostatní diskrétní součástky. Další možností je *evoluce na funkční úrovni* - využívají se složitější elementy, např. sčítačky, násobičky apod. Ukazuje se, že na úrovni hradel je návrh složitější a časově náročnější [14] než na úrovni větších stavebních bloků. Z tohoto pohledu je vždy nutné zvážit, z jakých elementárních prvků bude navrhovaný systém tvořen.

Rekonfigurovatelný obvod. Rekonfigurovatelný obvod může být v podstatě dvojího druhu. Máme možnost využít *komerčně dostupné zařízení* nebo si nechat vyrobit pro účely evoluce specifický *zákaznický obvod*. V dnešní době existují na trhu speciální čipy [16], které mají možnost být rekonfigurovány. Jedním z nich je např. rekonfigurovatelné pole tranzistorů FPTA (Field Programmable Transistor Array), které bylo navrženo primárně pro využití v oblasti evolučního návrhu. Jinou možností je využít hradlových polí FPGA, které již ze své podstaty umožňují rekonfiguraci. FPGA se skládají z mnoha elementů, které se propojují podle konfiguračního řetězce, který může mít i několik mega bytů. Při evoluci se využívá jen část konfiguračního řetězce, případně se uvnitř FPGA vytvoří tzv. virtuální rekonfigurovatelný obvod [18]. Druhý postup je v praxi běžnější, neboť novější typy FPGA se mohou díky chybnému konfiguračnímu řetězci zničit. Kromě toho je pro řadu aplikací doba rekonfigurace příliš dlouhá. Na poli komerčně dostupných obvodů existuje kromě číslicových FPGA i analogový rekonfigurovatelný obvod FPAA (Field Programmable Analog Array) [33], uvnitř kterého lze vytvořit téměř libovolný analogový obvod.

Výpočet fitness. Z hlediska výpočtu fitness rozlišujeme mezi *offline* a *online* vyvíjejícím se obvodem. V prvním případě (offline EHW) je evoluce simulována softwarově, pouze nejlepší chromozom se zapisuje do hardware. Tento přístup je někdy též označován jako *extrinsická evoluce*. Druhý přístup je zcela opačný, každý chromozom, který během procesu evoluce vznikne, se nahrává do rekonfigurovatelného obvodu a pomocí něj se vyhodnocuje. Na vstup rekonfigurovatelného obvodu jsou přikládána vstupní data a vyhodnocuje se získaná odezva obvodu. Tato metoda je nazývána *intrinsická evoluce*. Na první pohled se může zdát, že v obou uvedených přístupech není rozdíl (samozřejmě až na rychlost evoluce). Ukazuje se však, že intrinsická evoluce může přinést zajímavější výsledky. Evoluce využívající hardware je narozdíl od simulace schopna využít některých fyzikálních vlastností materiálu (pokud to architektura rekonfigurovatelného obvodu dovoluje). Musíme si však uvědomit skutečnost, že pokud evoluce využila pro realizaci obvodu některou vlastnost čipu, nemusí tuto vlastnost splňovat jiný čip i když se jedná o typově stejný hardware.

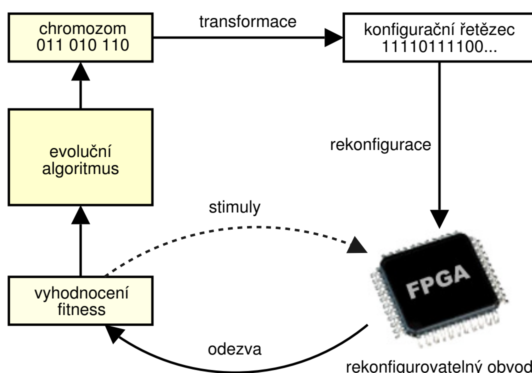
Umístění evolučního algoritmu. Z pohledu umístění evolučního algoritmu lze rozeznat tři druhy evoluce. Pokud je evoluční algoritmus provozován na samostatném procesoru odděleně od rekonfigurovatelné součástky, jedná se o evoluci *mimo čip* (*off-chip*). Typickým příkladem evoluce mimo čip je evoluční algoritmus běžící na běžném počítači, který komunikuje s rekonfigurovatelným obvodem. V případě, že evoluční algoritmus běží na procesoru, který je součástí vyvíjející se platformy, jedná se o evoluci *na čipu* (*on-chip*). V tomto případě je evoluční algoritmus typicky realizován v DSP[1] procesoru, který komunikuje s rekonfigurovatelným obvodem. Posledním typem je *kompletní hardwarová evoluce*, kdy je evoluční algoritmus implementován ve speciálním hardware, tzn. není využit procesor, který je řízen pomocí instrukcí. Kompletní hw realizace představuje nejrychlejší metodu z uvedených přístupů.

Fitness funkce. Fitness funkci rozlišujeme *statickou* a *dynamickou*. V prvním případě evoluce v určitém okamžiku končí a získaný obvod se nasadí do reálné aplikace. Během provozu obvodu se již evoluce nepoužívá. Tento způsob odpovídá dříve zmiňovanému evolučnímu návrhu obvodů. Cílem je získat inovativní řešení, doba evoluce je nepodstatná. Naopak je-li evoluce využívána během celého životního cyklu obvodu, jedná se o pravý vyvíjející se obvod. Cílem evoluce je zajistit co nejvyšší průměrný výkon, nikoliv nalézt inovativní řešení. Jednou z aplikací tohoto typu je obvod pro kompresi dat. Narozdíl od předchozího typu doba evoluce hraje velmi významnou roli, neboť je nutné dodržet maximální dobu odezvy [5].

[1] Digital Signal Processor - svou architekturou speciální procesor určený pro zpracování signálů

5 Obvodová realizace vyvíjejících se obvodů

Jak jsme již uvedli dříve, základem extrinsické evoluce je softwarový simulátor, který je schopen ohodnotit kvalitu obvodu získaného procesem evoluce prostřednictvím simulace jeho chování. Jedním z přínosů intrinsické evoluce je rychlejší vyhodnocení kvality obvodu, neboť simulace je ve většině případů časově náročnější než vyhodnocení fyzicky existujícího obvodu. Na druhou stranu je ale zapotřebí oproti extrinsické evoluci provést transformaci chromozomu a rekonfiguraci obvodu. Jednotlivé kroky evoluce jsou znázorněny na obrázku 9. U evoluce na čipu je většinou zapotřebí chromozom získaný evolučním procesem interpretovat a transformovat na konfigurační řetězec, který se nahraje do rekonfigurovatelného obvodu. Je nutné si uvědomit, že doba trvání rekonfigurace je závislá na použité platformě a ne vždy je zanedbatelná. Po rekonfiguraci následuje vyhodnocení reakce získaného obvodu na vstupní signály tzv. stimuly. Na základě reakce obvodu na všechny stimuly se stejným způsobem jako v případě extrinsické evoluce vypočítá fitness hodnota, která musí být pro kvalitnější obvody (z hlediska požadavků) vyšší.



Obrázek 9: Schema intrinsické evoluce

5.1 Kritéria pro úspěšnou platformu

Před několika lety bylo zformulováno několik kritérií [23], které musí splňovat evoluční platforma vhodná pro intrinsickou evoluci. Z našeho pohledu nejpodstatnější požadavky jsou uvedeny v následujících odstavcích.

Neomezený počet rekonfigurací. Na trhu sice existuje mnoho programovatelných obvodů, ne všechny jsou však vhodné pro intrinsickou evoluci. Některé programovatelné obvody jsou koncipovány tak, že umožňují pouze jedno naprogramování. Jiné obvody je možné přeprogramovat několikrát, ve většině případů však snesou pouze nízký počet programování, což je dáno tím, že konfigurace je uložena v paměti EEPROM. Opakovaná rekonfigurace vede ke zničení obvodu. Evoluční experimenty však potřebují miliony vyhodnocení, z toho důvodu je nutné, aby bylo možné obvod rekonfigurovat v nejlepším případě libovolněkrát. Abychom splnili tento požadavek, musíme volit takové programovatelné obvody, v nichž je konfigurační řetězec uložen v paměti RAM.

Rychlá případně částečná rekonfigurace. Jestliže jsou potřeba miliony vyhodnocení, měl by být proces vyhodnocení i rekonfigurace rychlý. Moderní programovatelné obvody ale mají mnoho konfigurovatelných elementů a délka konfiguračního řetězce roste s počtem elementů. Rekonfigurace obvodu může proto představovat slabé místo evolučního procesu. Jednou z možností, jak zmenšit dobu potřebnou pro rekonfiguraci, je použít programovatelné obvody obsahující vysokorychlostní konfigurační porty. Jinou alternativou může být snaha o vyhodnocení více jedinců současně. Tento přístup bohužel přináší daň za rychlost. Není možné pracovat s velkou populací a s jakýmkoliv evolučním algoritmem. Elegantnější řešení spočívá v částečné rekonfiguraci, kdy jsou do zařízení nahrávány pouze změny. Nicméně ze strany rekonfigurovatelného obvodu musí být částečná rekonfigurace podporována. Výhodou je, že tato metoda podobně jako metoda využívající vysokorychlostních portů neklade žádné omezení na evoluční algoritmus.

Nezničitelnost případně schopnost kontroly. Pokud spojíme dva výstupy z nichž každý má jinou napěťovou úroveň, vznikne, v případě běžně používané CMOS techologie, zkrat. Proud, který tímto spojem teče může vést a zpravidla vede ke zničení části v horším případě celého obvodu. Většina dnešních komerčních hardwarových platforem FPGA je navržena tak, že ne všechny kombinace jedniček a nul jsou v konfiguračním řetězci přípustné. Je však možné se s tímto nedostatkem vypořádat. První možností je vystavět nad touto architekturou abstraktnější architekturu, ve které ke zmíněnému efektu nedojde. V rekonfigurovatelném obvodu vznikne nový tzv. virtuální rekonfigurovatelný obvod [18]. Jinou cestou je před vlastní konfigurací zkontrolovat, zda vzniklý řetězec je platný a nezpůsobí při naprogramování v hardware zkrat, což nemusí být vůbec triviální úloha.

Jemná granularita rekonfigurace. Abychom dali možnost evoluci nacházet inovativní řešení, musí být evoluce schopna pracovat na nízké úrovni abstrakce. Nejzajímavější obvody můžeme nalézt pouze na velmi nízké úrovni – např. na úrovni hradel nebo tranzistorů. Z tohoto důvodu musí dobrá platforma dovolovat jemnou konfiguraci. Většina dnešních rekonfigurovatelných obvodů je založena na opakujících se větších blocích, např. FPGA se skládájí z konfigurovatelných bloků CFB. Z principu FPGA již ale víme, že jednotlivé bloky lze konfigurovat na jemnější úrovni. Moderní rekonfigurovatelné obvody FPGA tedy tuto podmínku bezesporu splňují.

5.2 Analogové obvody

V analogové oblasti je známo poměrně málo rekonfigurovatelných obvodů, neboť značné úsilí se soustředí na digitální obvody.

Jediným komerčně dostupným analogovým rekonfigurovatelným obvodem je obvod FPAA (Field Programmable Analog Array) firmy Anadigm [33]. Jedná se vpodstatě o ekvivalent klasických logických programovatelných obvodů FPGA. FPAA dává návrháři možnost (podobně jako FPGA) popsat obvod na vyšší úrovni abstrakce, aniž by musel přemýšlet na úrovni strukturních prvků jako jsou operační zesilovače, kondenzátory, odpory apod. Proces návrhu se stává jednodušším a komplikované obvody, které by bylo nutné navrhovat týdny nebo mě-

síce, dokáže vytvářet v krátkém čase i návrhář, který se na analogové obvody nespecializuje. Dynamická rekonfigurace navíc umožňuje kdykoliv provést změnu designu. Tento obvod je možné použít pro samokalibrující se systémy, adaptivní řízení, adaptivní filtraci apod. Jelikož je konfigurace uchovávána v paměti RAM, je tato platforma vhodná i pro účely evoluce.

Druhým nejznámějším analogovým rekonfigurovatelným obvodem je FPTA (Field Programmable Transistor Array). Jedná se o zákaznický obvod ASIC, který byl vyvinut především pro účely evolučního návrhu. Rekonfigurovatelný obvod je sestaven z 256 buněk, které jsou umístěny v matici 16×16. Jednotlivé buňky jsou vzájemně propojeny do všech osmi směrů, polovina buněk se skládá z tranzistorů PMOS a druhá polovina z NMOS. Každá buňka obsahuje celkem 14 tranzistorů, několik diskrétních součástek a 74 konfigurovatelných propojek, které určují nejen propojení tranzistorů uvnitř buňky ale i propojení buněk navzájem.

5.3 Číslicové obvody

Vyvíjející se obvod máme možnost vystavět na třech různých platformách. Nejčastěji používanou platformou jsou snadno dostupné komerční rekonfigurovatelné obvody FPGA, které poskytují dostatečné množství prostředků jak pro extrinsickou tak i pro intrinsickou evoluci. Jinou možností je využít některého z komerčně vyráběných EHW čipů nebo si nechat vyrobit zákaznický rekonfigurovatelný obvod ASIC. Je zřejmé, že poslední dvě varianty jsou však silně aplikačně specifické obvody, které jsou velmi nákladné a představují řešení, které nemusí být dostupné pro každého.

5.3.1 Specifické EHW obvody

Protože EHW čipy se zatím nedočkaly příliš velkého rozšíření, je možné je považovat za zákaznické ASIC obvody. Doposud jsou známy čtyři druhy EHW čipů [5], které jsou určeny pro průmyslové aplikace. Prvním z nich je analogový čip určený pro mobilní telefony. Ačkoliv snahou je nahradit veškeré analogové obvody digitálními, existují aplikace, kde je zapotřebí vysokorychlostních analogových obvodů (např. komunikace). V mobilních telefonech jsou zapotřebí filtry, kde i 1% odchylka od střední frekvence je neakceptovatelná. Protože výroba takto přesných analogových filtrů je ale velmi náročná a drahá, byl vytvořen čip, který umožňuje pomocí několika konfigurovatelných elementů doladit nepřesnosti ve frekvenci. EHW čip se skládá z analogového obvodu, jehož parametry lze modifikovat a pomocí genetického algoritmu doladit tak, aby i filtry, které nesplňovaly specifikaci, mohly být použity.

Dalším typem je EHW čip určený pro kompresi dat. Tento obvod je určen pro nasazení v oblasti elektrofotografického tisku, který dovoluje tisk knih v kvalitě fotografie. Základním problémem je objem dat a nedostatečná přenosová rychlost. Kniha o 100 stranách představuje asi 7GB obrazových dat, které musí být přeneseny rychlostí 1800MB/min. Aby bylo možné tento přenos realizovat, je zapotřebí využít velmi efektivní, rychlé a přitom kvalitní komprese dat. To jsou ale obecně protichůdné požadavky, neboť kvalitní komprese je většinou časově velmi náročná. Proto byl vytvořen čip, jehož úkolem je realizovat takto náročnou kompresi. Jedná se o pravý vyvíjející se obvod obsahující evoluční algoritmus, který má zajistit co nejvyšší průměrný výkon a kvalitu. Z různých experimentů se ukazuje, že pomocí evoluce lze dosáhnout lepšího kompresního poměru, než pomocí dostupných kompresních metod.

Posledním z existujících čipů je čip určený pro adaptivní řízení. Použití extrinsické evoluce, která vyžaduje pro svůj provoz osobní počítač, může být problematické v případě, že potřebujeme obvod, který je lehký a malý. Proto vznikl tento čip, který integruje evoluční algoritmus a rekonfigurovatelnou logiku. EHW čip byl použit pro autonomní řízení mobilního robota a pro řízení protetické končetiny, což představuje v této oblasti výrazný pokrok. Pomocí evoluce se učí umělá ruka reagovat na elektrické signály vysílané člověkem a nikoliv naopak, jak tomu bylo doposud. Čip se skládá z programovatelného logického pole PLA, genetického algoritmu a 16bitového procesoru.

5.3.2 Využití FPGA

První experiment byl úspěšně proveden s využitím FPGA čipu XC6126 v roce 1996 [24]. Evoluce byla prováděna přímo nad konfiguračním řetězcem bez nutnosti jakékoliv transformace a bylo využito 10×10 bloků FPGA čipu, čímž bylo evoluci umožněno pracovat na nejnižší možné úrovni abstrakce. Cílem bylo navrhnout obvod, který umí rozlišit dva signály o frekvenci 1kHz a 10kHz. Fitness hodnota byla počítána podle odezvy obvodu na 500ms trvající signál složený z libovolně dlouho trvajících úseků jednotlivých signálů. Nebyl použit žádný hodinový signál k synchronizaci, obvod měl pouze jediný vstup, na který se přiváděl signál. Pomocí evoluce se podařilo takto specifikovaný obvod nalézt a dokonce s menším počtem elementů, než by použil návrhář. Obvod však byl velmi zvláštní a jeho funkci se nepodařilo zcela popsat.

Dnešní FPGA obvody obsahují velmi mnoho konfigurovatelných elementů, jejich konfigurační řetězec je velmi dlouhý a jeho struktura je většinou komplikovaná, navíc některé kombinace mohou vést ke zničení celého obvodu. Přímá evoluce celého konfiguračního řetězce tedy vůbec nepřichází v úvahu. Transformace z chromozomu na konfigurační řetězec, případně jeho kontrola se stává také velmi obtížnou a časově náročnou. Určitým východiskem je evoluce části konfiguračního řetězce. Situaci ovšem komplikuje doba potřebná pro rekonfiguraci, která i v případě částečné rekonfigurace je dosti velká. Toto tvrzení samozřejmě platí pouze v případě, že evoluci v hardware provádíme především proto, abychom výpočet zrychlili. Pokusů o přímou rekonfiguraci FPGA, nebo s využitím částečné rekonfigurace, kterou dnešní FPGA obvody podporují, bylo provedeno několik [26], avšak výsledky nebyly příliš překvapivé.

Jako nejvýhodnější se jeví vytvořit uvnitř FPGA virtuální rekonfigurovatelný obvod, který všechny nedostatky řeší. O tomto přístupu se zmiňuje následující kapitola.

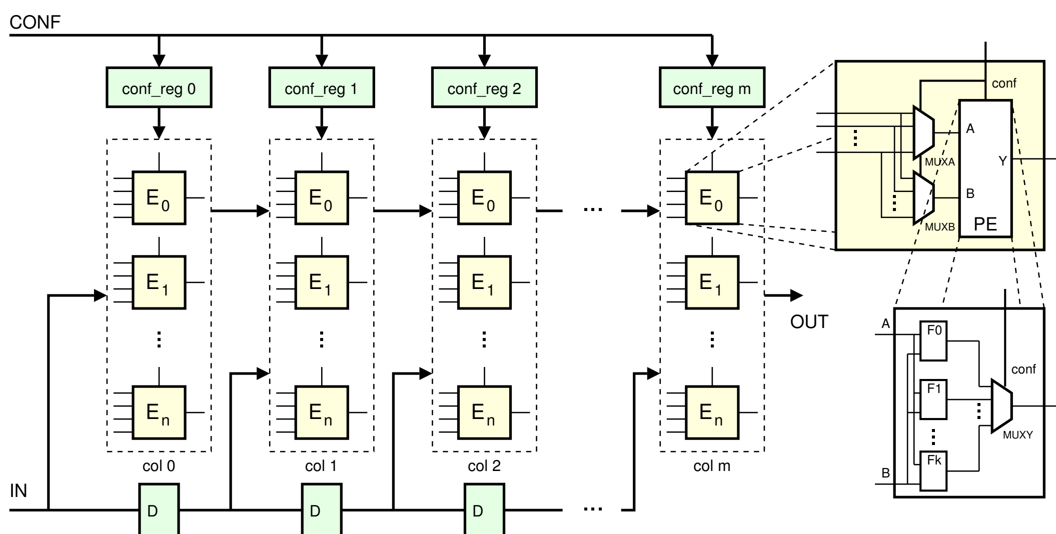
5.3.3 VRC

Princip virtuálního rekonfigurovatelného obvodu VRC (Virtual Reconfigurable Circuit) byl představen v [18] a uveden jako druh velmi rychle rekonfigurovatelné platformy. VRC je běžný číslicový obvod, který vznikne uvnitř FPGA po nahrání příslušného designu. Protože konfigurace VRC je uložena v registrech, je možné jej libovolněkrát a velmi rychle rekonfigurovat.

Princip VRC vychází z kartézského genetického programování, které jsme uvedli v kapitole 2.5. Obvod se skládá z matice programovatelných elementů PE, programovatelné propojovací sítě, konfigurační paměti a konfiguračního portu. Jedná se vlastně o další rekonfigurovatelnou vrstvu vystavěnou nad FPGA, jejímž primárním cílem je zrychlit rekonfiguraci. Výhodou je, že máme možnost vytvořit programovatelné elementy s ohledem na aplikaci. Budeme-li např.

realizovat filtr, mohou být jednotlivé elementy tvořeny násobičkami, budeme-li chtít realizovat logický obvod, budou tvořeny logickými funkcemi apod.

Struktura virtuálního rekonfigurovatelného obvodu je znázorněna na obrázku 10. Rekonfigurovatelný obvod je složen z maticově uspořádaných základních elementů E. Každý element obsahuje konfigurační port, určitý počet vstupů a jeden výstup. První část vstupů je napojena na globální vstup IN, druhá část vstupů je napojena na výstupy elementů v předchozím sloupci. Konfiguraci pro všechny elementy i-tého sloupce uchovává konfigurační registr conf_reg i.



Obrázek 10: Virtuální rekonfigurovatelný obvod

Uvnitř každého elementu E je programovatelný element PE, který může realizovat jednu z k pevně daných funkcí. PE element uvedený na obrázku má dva vstupy, na každý z nich lze pomocí multiplexorů připojit jeden ze vstupů elementu E. Element PE je složen z k dvouvstupových funkčních bloků, které pracují paralelně. Výběr jedné z k funkcí se provádí opět pomocí multiplexoru.

Vyhodnocení odezvy VRC obvodu je možné provést buď asynchronně nebo synchronně. V případě synchronního řízení je možné obvod provozovat v zřetězeném provozu. Na vyhodnocení jedné vstupní kombinace pak potřebujeme pouze jeden hodinový takt.

6 Číslicová filtrace

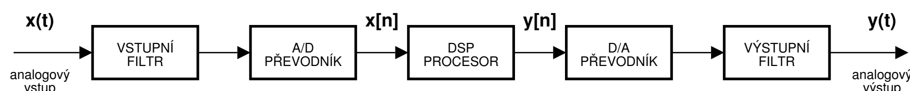
Jelikož budeme potřebovat alespoň základní znalosti z oblasti číslicové filtrace, věnuje se tato kapitola problematice číslicového zpracování signálů.

Digitální nebo též číslicové zpracování signálů hraje v dnešní době velmi důležitou roli a je využíváno v mnoha různých aplikacích. Aniž bychom si to uvědomovali, využíváme každodenně systémy, které zpracovávají signály číslicově. Velmi složité číslicové zpracování se používá např. v běžných mobilních telefonech. Jedná se o různé digitální kodéry, které zajišťují kompresi, kódování řeči apod. Hlavními oblastmi, kde se číslicové zpracování signálu využívá jsou: zpracování audio signálu, zpracování řečového signálu, telekomunikace, ale i zpracování obrazu. Běžným cílem je zlepšit kvalitu signálu (např. potlačit případně odstranit šum), extrahovat ze signálu důležité informace nebo oddělit dva a více signálů od sebe (např. telekomunikace).

Typickým postupem je převést spojitý analogový signál na digitální diskrétní signál, ten většinou pomocí hardware zpracovat a po zpracování převést zpět na analogový signál. Na první pohled se může zdát, že jsme tímto postupem zpracování analogového signálu velmi zkomplikovali, avšak opak je pravdou. Tento postup má řadu výhod. Digitální zpracování je většinou realizováno pomocí DSP procesorů (Digital Signal Processor), což jsou procesory určené pro práci se signálem, které akcelerují nejčastěji používané operace. Výhodou je, že pouhou změnou programu můžeme zpracovávat signál jiným např. efektivnějším způsobem. To u klasických analogových součástek není možné, každá změna ve zpracování má za následek výměnu celé desky. Na stejném hardware je možné realizovat mnoho různých úloh, v určitých aplikacích je možno dosáhnout vyšší přesnosti oproti analogové verzi s nižšími náklady. Další výhodou je neměnnost funkčnosti - nedochází k žádnému stárnutí součástek a tím k rozlaďování obvodu. Díky digitálnímu zpracování můžeme dosahovat charakteristik, které jsou klasickými analogovými filtry nedostupné.

Kromě výhod má však digitální zpracování i některé nevýhody. Oproti analogovému zpracování jsme limitováni konečnou přesností výpočtu. Dalším omezujícím faktorem může být hranice maximální rychlosti zpracování signálu.

Zjednodušený blokový diagram zpracování analogového signálu můžeme vidět na obrázku 11. Spojitý analogový signál $x(t)$ je nutné před vlastním převodem na digitální vzorky $x[n]$



Obrázek 11: Blokový diagram digitálního zpracování analogového signálu

modifikovat pomocí vstupního filtru. Vstupní filtr je obyčejná dolní propust, která nepropustí frekvence vyšší, než je polovina vzorkovací frekvence. Toto předzpracování je nutné z důvodu splnění vzorkovacího teorému a následné správné rekonstrukce [6]. Po vstupní filtraci se převádí spojitý signál na vzorkovaný pomocí analogově-digitálního převodníku. Zde se vnáší první nepřesnosti do zpracování signálu, neboť nejsme schopni v nekonečně malém čase odečíst přesně hodnotu analogového signálu $s(t)$. Digitální vzorky jsou ve většině případů zpracovávány po-

mocí specializovaných procesorů DSP. Po číslicovém zpracování je nutné pomocí digitálně-analogového převodníku převést diskrétní vzorky zpět na spojitou hodnotu. Pokud bychom využívali přímo signál z výstupu digitálně-analogového převodníku signál, zjistili bychom, že nemá spojitý průběh a obsahuje velké množství harmonických frekvencí. Z toho důvodu se ještě řadí na výstup stejná dolní propust, jaká je na vstupu. Hlavní funkcí výstupního filtru je interpolovat hodnoty mezi vzorky a vyhladit výsledný signál, aby měl spojitý průběh.

Základem číslicového zpracování jsou číslicové filtry. Pod pojmem filtr si můžeme představit systém, který má obecně několik vstupů a jeden nebo více výstupů. Pokud budeme uvažovat nejjednodušší filtr, který má jeden vstup a jeden výstup, pak vstupem filtru je spojitý signál $x(t)$ a výstupem signál $y(t)$. Úkolem filtru je určitým způsobem ovlivnit vlastnosti signálů $x(t)$ tak, abychom dosáhli požadovaného výsledku $y(t)$. Může se např. jednat o změnu tvaru signálu, změnu frekvenční charakteristiky signálu nebo změnu fázové charakteristiky. Typickou operací filtrů může být např. realizace dolní propusti, zesilovacího článku, rezonátoru, ale i derivace, výpočtu průměru, apod.

Číslicové filtry lze rozdělit na dvě velké oblasti a sice filtry lineární a filtry nelineární. Na oba typy filtrů se podíváme z pohledu jednorozměrného signálu (kapitola 6.1) a dvourozměrného signálu (kapitola 6.2). S jednorozměrným signálem se pracuje především v oblasti zpracování zvuku, s dvourozměrným v oblasti zpracování obrazu. Dvourozměrný signál je z hlediska zpracování pouhým zobecněním jednorozměrného signálu, proto se teoretické části budeme více věnovat pouze u jednorozměrného signálu.

6.1 Filtrace jednorozměrného signálu

6.1.1 Lineární číslicové filtry

Lineární číslicové filtry jsou takové filtry, u kterých je zaručena linearita. Reaguje-li filtr na vstupní signál $x_1[n]$ výstupem $y_1[n]$ ($x_1[n] \rightarrow y_1[n]$) a na jiný vstupní signál $x_2[n]$ výstupem $y_2[n]$ ($x_2[n] \rightarrow y_2[n]$), pak je tento filtr lineární právě tehdy, když je splněna aditivita a homogenita. Obě podmínky lze vyjádřit následujícím předpisem:

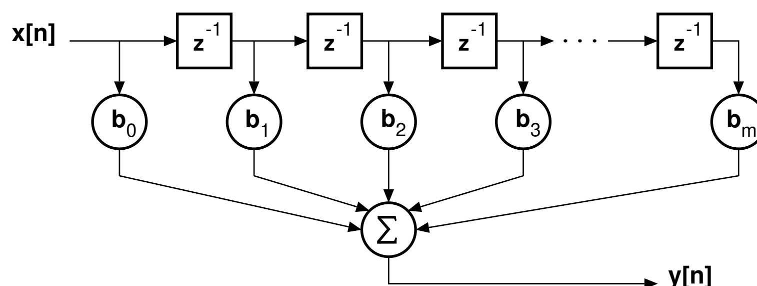
$$a_1x_1[n] + a_2x_2[n] \rightarrow a_1y_1[n] + a_2y_2[n], \tag{6.1.1}$$

Pokud kromě linearity zaručíme i časovou invariantnost filtru, je možné odvodit reakci filtru na libovolný vstupní signál, aniž bychom museli provádět simulaci. Abychom mohli definovat chování filtru, je nutné zjistit impulsní odezvu $h[n]$, tzn. jakým způsobem filtr reaguje na jednotkový impuls.

Odezvu filtru na obecný vstupní signál nyní můžeme vypočítat tak, že vstupní signál na jednotkové impulsy rozložíme a s každým jednotkovým impulsem spustíme jednu impulsní odezvu. Nutno dodat, že každá spuštěná impulsní odezva musí být vynásobena amplitudou příslušejícího vstupního vzorku. Abychom získali výsledný signál, musíme všechny v čase posunuté odezvy sečíst. Vynásobením amplitudou ani součtem jednotlivých odezev jsme neporušili podmínku linearity. Tato operace se nazývá konvoluce a je definována pomocí následujícího vztahu:

$$y[n] = h \star x[n] = \sum_{k=0}^{M-1} h[k]x[n-k] \tag{6.1.2}$$

Vyjdeme-li ze znalosti, že konvoluční rovnice 6.1.2 plně popisuje operaci filtrace, je zřejmé, že se filtry v podstatě skládají z bloků realizující násobení konstantou a jednoho bloku realizujícího sčítání. Z hlediska hardwarové implementace se jedná o poměrně jednoduše implementovatelné operace. Musíme si však uvědomit, že tzv. filtrovací okno, které vymezuje impulsní odezva filtru $h[k]$, se po vstupním signálu neustále posouvá, neboť výsledkem konvoluce je pouze jedna hodnota $y[n]$ příslušející n-tému vzorku výstupu. Podrobnějším studiem bychom zjistili, že není zapotřebí posouvat filtrovací okno, ale lze posouvat vstupní signál. K tomuto účelu slouží tzv. zpožďovací linka, která je tak dlouhá, jaký je počet hodnot impulsní odezvy. Schéma tohoto filtru je na obrázku 12.

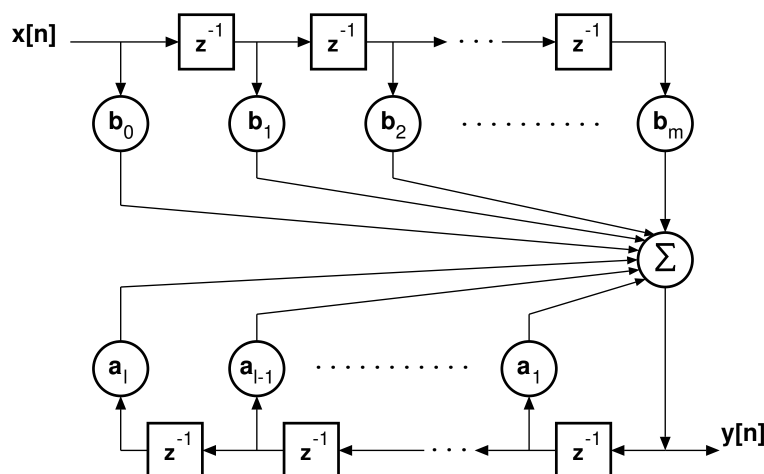


Obrázek 12: Schéma FIR filtru

Rozšíříme-li rovnici 6.1.2 o zpětnou vazbu, získáme obecný filtr, který lze popsat touto rovnicí:

$$y[n] = \sum_{k=0}^{M-1} b_k x[n-k] + \sum_{k=1}^{N-1} a_k y[n-k] \tag{6.1.3}$$

Výstupní vzorek $y[n]$ se odvozuje jednak z aktuálního vstupního vzorku $x[n]$ a jeho zpožděných variant a dále ze zpožděných variant výstupního vzorku. Implementace je znázorněna na obrázku 13.



Obrázek 13: Obecný rekurzivní IIR filtr

Podle délky trvání impulsní odezvy lze rozdělit číslicové filtry na dva základní druhy. Filtry s konečnou impulsní odezvou – FIR (Finite Impulse Response) a filtry s nekonečnou impulsní odezvou – IIR (Infinite Impulse Response). Nekonečnost impulsní odezvy filtrů IIR je způsobena tím, že jsme vytvořili zpětnou vazbu z výstupu na vstup.

Vyjdeme-li z rovnice 6.1.3 definující činnost filtru, a položíme-li vstupní vzorek $x[1] = 1$ a všechny ostatní $x[i] = 0$, pak $y[1] = 1$. Další vzorky výstupu $y[n]$ se již odvozují pouze z předešlých hodnot výstupu protože na vstupu bude stále hodnota 0. Pokud by měl filtr pouze jeden koeficient zpětné vazby $a_0 \in (0, 1)$, pak pro n-tou hodnotu výstupu musí platit $y[n] = {a_0}^n y[1]$. Protože $y[1] = 1$, je hodnota $y[n]$ rovna nule pouze v nekonečnu, všechny ostatní koeficienty jsou nenulové. Jedná se tedy o rekurzivní IIR filtr.

Filtry typu FIR mají několik výhod. Pokud bychom provedli z-transformaci rovnice 6.1.3, zjistili bychom, že tyto filtry jsou vždy stabilní. Realizujeme-li filtr přesně, jak je uvedeno na obrázku 12, bude jeho stabilita zaručena. Existuje totiž možnost realizovat filtry FIR pomocí tzv. IIR rezonátorů.

Další pozitivní vlastností je, že FIR filtry mají lineární zpoždění, tzn. fáze jednotlivých frekvenčních složek vstupního signálu zůstávají zachovány. Tato vlastnost je požadována v mnoha aplikacích – např. při přenosu dat, v biomedicíně, při zpracování audio signálu a video signálu. Fázová charakteristika IIR filtrů nemusí být lineární, zvláště na okrajích pásem.

Nevýhodou oproti rekurzivním variantám je velmi velký počet koeficientů filtru a tedy vyšší nároky na paměť vstupních vzorků a na čas zpracování. To lze demonstrovat na následujícím příkladu. Představme si, že cílem je navrhnout filtr, který realizuje dolní propust, jejíž charakteristika je v oblasti přechodu velmi strmá (blíží se obdélníkovému průběhu). Pokud bychom navrhli FIR filtr, který splňuje tyto požadavky, zjistíme, že je zapotřebí např. 500 koeficientů. V případě IIR filtru by však bylo zapotřebí pouze 10 koeficientů. Je tedy v praxi nutné vždy zvážit, jaký typ filtru zvolit.

Nežádoucí efekt vznikající při použití limitovaného počtu bitů v implementaci FIR filtru, zaokrouhlovací šum a kvantizační chyba je mnohem méně závažný, než u filtrů IIR a často se zanedbává.

Zavedením zpětné vazby se získá mnoho výhod, ale je nutné počítat s tím, že můžeme vytvořit nestabilní filtr. V případě filtru, který je na mezi stability může vlivem kvantizační chyby dojít k jeho rozkmitání. Na tyto problémy je třeba pamatovat při návrhu a promítnout do něj i vliv chyb.

Výhodou IIR filtrů je, že mohou vzniknout přímou transformací analogových filtrů. Návrh FIR filtrů je algebraicky složitější.

Výběr mezi realizací pomocí IIR nebo FIR se v zásadě provádí podle toho, zda požadujeme strmé hrany ve frekvenční charakteristice nebo malý počet koeficientů z hardwarových důvodů (IIR) nebo zda požadujeme lineární fázi, případně počet koeficientů není příliš velký (FIR).

6.1.2 Nelineární číslicové filtry

Někdy se stává, že není možné vytvořit filtr s požadovanou charakteristikou, který by byl lineární. K tomu dochází zejména tehdy, pokud šum není aditivní, nebo nemá gausovský charakter. Je známo [6], že lineární filtry mohou odstranit aditivní šum o vyšší frekvenci jen tehdy, pokud se spektrum signálu a šumu ve frekvenční doméně nepřekrývá. Při zpracování dvoudimenzio-

nálního signálu (např. obrazu) signál obsahuje ve většině případů užitečnou vysokofrekvenční složku jako jsou hrany případně jemnější detaily, kterou bychom si přáli ponechat. V případě, že na takový signál použijeme lineární dolní propust, bude výsledný obraz velmi rozmazaný. Právě z tohoto důvodu se používají nelineární filtry.

Ve zpracování hudby můžeme nalézt tři hlavní typy aplikací, ve kterých se nelineárního zpracování využívá [22]. Do první kategorie patří aplikace, jejichž cílem je dynamicky řídit tvar případně charakter signálu pomocí několika variabilních parametrů. Druhá třída je navržena s cílem deformovat vstupní signál – tzv. distorze. Procesory, které umí provádět různé druhy deformací, se používají např. při zpracování signálu z elektrických kytar pro navození různých efektů. Třetí kategorii reprezentují zařízení běžně označované jako excitery a enhancery. Hlavním úkolem těchto filtrů je přidat další harmonické do signálu, čímž se dosahuje velmi jemného vylepšení charakteristiky zvuku.

6.2 Filtrace obrazu

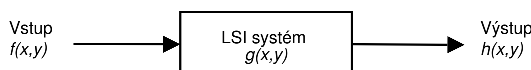
Obraz pořízený kamerou či jiným snímacím zařízením se málo kdy používá přímo bez dalšího předzpracování. Obraz může být znehodnocen rozdíly v intenzitě (šum), barevnou vadou nebo špatným kontrastem.

Tato kapitola ukazuje několik metod číslicové filtrace, které se používají pro zpracování obrazů. Omezíme se pouze na zpracování obrazů v odstínech šedi, barevný obraz je pouhým zobecněním, neboť ve většině případů je obraz zpracováván separátně po jednotlivých barevných kanálech. Detailněji se touto problematikou zabývá např. literatura [17].

Ve zpracování obrazu se nejvíce používají dvourozměrné FIR filtry a pracuje se s malými okny 3×3, 5×5 nebo 9×9.

6.2.1 Lineární filtry

Podobně jako u jednorozměrného signálu může být filtrace modelována lineárním systémem, který je prostorově invariantní – LSI system (Linear Space Invariat system). Prostorová invariance nám říká, že filtr bude vykazovat stejné vlastnosti v libovolném bodě obrazu. V praxi to znamená, že výpočet bude pro každý bod obrazu probíhat stejně.



Obrázek 14: LSI systém

LSI systém může být kompletně popsán jeho impulsní odezvou $g(x, y)$, jak ukazuje obrázek 14. Aby byl systém lineární, musí splňovat následující vztah:

$$a \cdot f_1(x, y) + b \cdot f_2(x, y) \rightarrow a \cdot h_1(x, y) + b \cdot h_2(x, y) \tag{6.2.4}$$

kde $f_1(x, y)$ a $f_2(x, y)$ jsou libovolné dva vstupní obrazy, $h_1(x, y)$ a $h_2(x, y)$ jsou obrazy korespondující s odezvou filtru na vstupy f_1 a f_2, a a b jsou libovolné konstanty.

Výstup $h[i, j]$ diskrétního systému, který splňuje výše uvedený vztah, je roven konvoluci vstupního obrazu $f[k, l]$ s impulsní odezvou $g[i, j]$.

$$h[i, j] = f[i, j] \star g[i, j] = \sum_{k=-\frac{n}{2}}^{\frac{n}{2}} \sum_{l=-\frac{m}{2}}^{\frac{m}{2}} f[k, l] g[i - k, j - l] \tag{6.2.5}$$

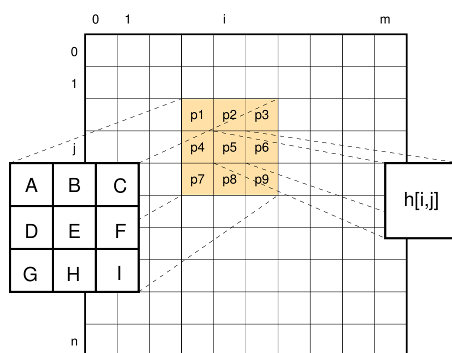
Pokud bychom prověřili definici konvoluce, zjistíme, že splňuje požadavek linearity, neboť

$$g[j, i] \star \{a_1 f_1[i, j] + a_2 f_2[i, j]\} = a_1\{g[i, j] \star f_1[i, j]\} + a_2\{g[i, j] \star f_2[i, j]\} \tag{6.2.6}$$

Konvoluce vlastně ve zpracování obrazu znamená součet vážených sum pixelů obrazu. Impulsní odezva $g[i, j]$ je běžně nazývána konvoluční maska. Pro každý pixel obrazu $[i, j]$ se hodnota výstupního pixelu $h[i, j]$ určí tak, že střed konvoluční masky posuneme do bodu $[i, j]$ a vypočítáme váženou sumu pixelů v okolí $[i, j]$, přičemž jednotlivé váhy odpovídají hodnotám

v konvoluční masce. Výpočet hodnoty pixelu v bodě $[i, j]$ je ilustrován obrázkem 15. Hodnota na výstupu $h[i, j]$ se vypočítá jako vážený součet

$$h[i, j] = Ap_1 + Bp_2 + Cp_3 + Dp_4 + Ep_5 + Fp_6 + Gp_7 + Hp_8 + Ip_9 \tag{6.2.7}$$



Obrázek 15: Výpočet odezvy h[i,j] s konvoluční maskou 3×3. Střed masky odpovídá hodnotě E, pozice v obrazu [i,j], váhy A,B,. . . , odpovídají hodnotám g[k,l]

Typy šumu

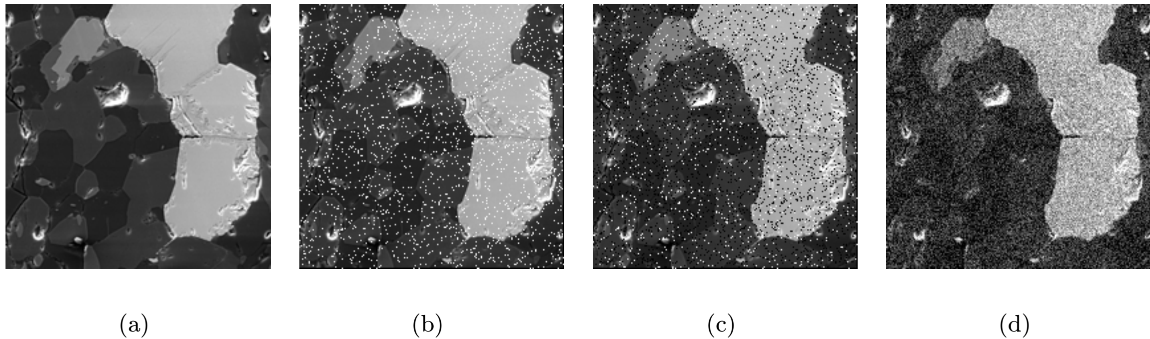
Jak jsme se zmínili dříve, obraz bývá často porušen náhodnými odchylkami v hodnotách intenzity. Tyto nežádoucí odchylky označujeme jako šum. Běžné typy šumu jsou šum typu *sůl a pepř*, *impulsní šum* a *gausovský šum*. Šum typu *sůl a pepř* obsahuje náhodné výskyty hodnot s nejnižší a nejvyšší intenzitou. Oproti tomu *impulsní šum* obsahuje pouze náhodné výskyty nejvyšší hodnot intenzity. *Gausovský šum* se od obou typů zcela odlišuje - obraz je poškozen hodnotami intenzity, které jsou náhodné. Vykazují však gausovské případně normální rozložení. Tento typ šumu je velmi dobrým modelem pro šum vznikající při snímání díky nedokonalosti senzorů. Příkladem může být šum v obraze, který vzniknul při snímáním CCD čipem.

Mean filter – průměr

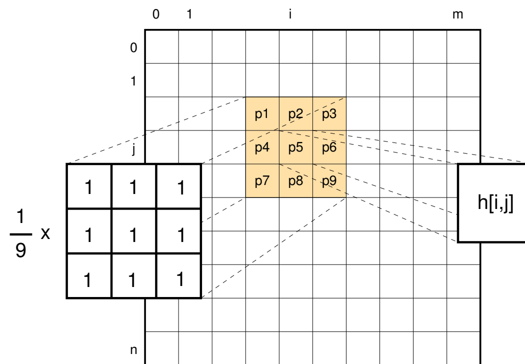
Jedním z nejjednodušších lineárních filtrů je filtr, který realizuje operaci lokálního průměru. Hodnota každého pixelu ve výstupním obrazu je nahrazena průměrem ze všech hodnot, které jsou v jeho okolí ve vstupním obraze. Filtr počítající průměr nad oknem 3×3 je dán následujícím předpisem:

$$h[i, j] = \frac{1}{9} \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} f[k, l] \tag{6.2.8}$$

Pokud porovnáme tuto rovnici s rovnicí 6.2.5, zjistíme, že všechny hodnoty konvoluční masky jsou shodné $g[i, j] = 1/9$. Konvoluci jsme vlastně zredukovali na operaci jejichž výsledkem je průměrná hodnota z daného okolí (lokální průměr). Velikost konvolučního okna značně ovlivňuje výslednou kvalitu. Pokud zvolíme okno velké, dostaneme obraz, který bude velmi rozostřen. V opačném případě zvolíme-li okno malé, nemusí dojít k výraznému zlepšení.



Obrázek 16: Ukázka obrazu znehodnoceného různým druhem šumu. Obraz a) originální b) poškozený impulsním šumem c) poškozený šumem typu pepř a sůl d) poškozený 10% gausovským šumem



Obrázek 17: Příklad ilustrující mean filtr požívající okolí 3x3

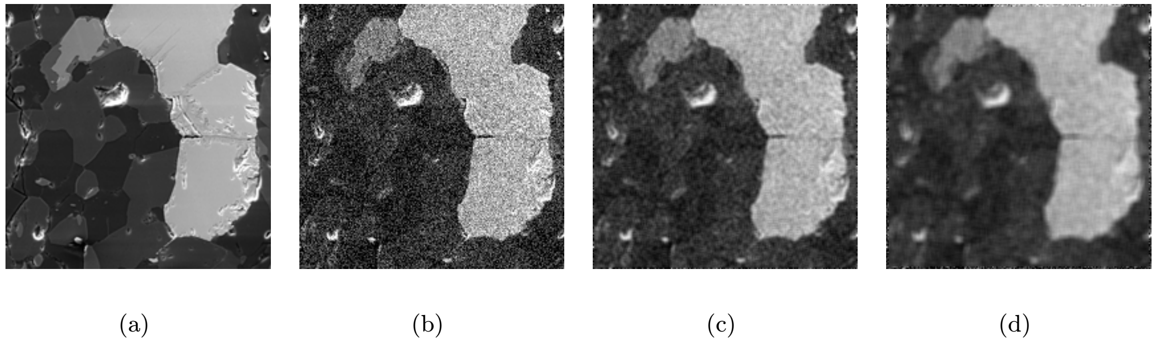
6.2.2 Nelineární filtry

Pokud filtr nesplňuje podmínku linearity, tzn. výstup není roven vážené sumě okolních pixelů, jedná se o nelineární filtry. Ty můžeme dále dělit podle toho, zda splňují podmínku prostorové invariance, či nikoliv.

Jedním z prostorově invariantních filtrů je třída zásobníkových filtrů (Stack Filters) [31]. Narozdíl od mnoha jiných nelineárních filtrů existuje k zásobníkovým filtrům marematický aparát [21], který umožňuje provádět jejich "frekvenční" rozbor podobně, jako tomu je u klasických lineárních filtrů, kde pomocí Fourierovi transformace jsme schopni určit chování filtru. Speciálním případem stack filtru je medián, tak jak ho známe. Jedná se o filtr, který se ve zpracování signálu používá běžně, neboť je snadno hardwarově implementovatelný.

Median filter – medián

Hlavní nevýhodou lokálně průměrujících filtrů je sklon k rozmazávání ostrých přechodů v obraze. Alternativním přístupem je nahrazení každého pixelu obrazu hodnotou mediánu z

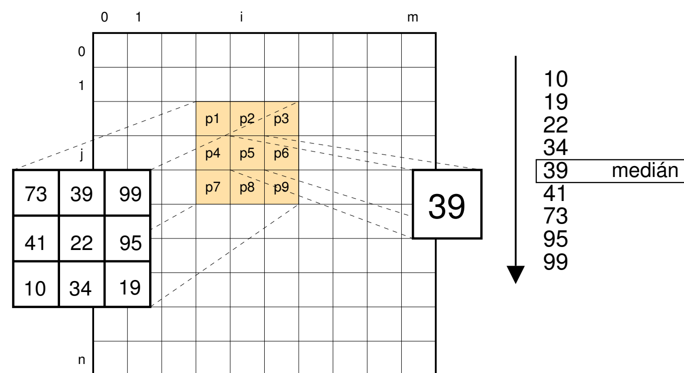


Obrázek 18: Ukázka filtrace obrazu znehodnoceného gausovským šumem. Obraz a) originální b) poškozený gausovským šumem c) filtrovaný maskou 3x3 d) filtrovaný maskou 5x5

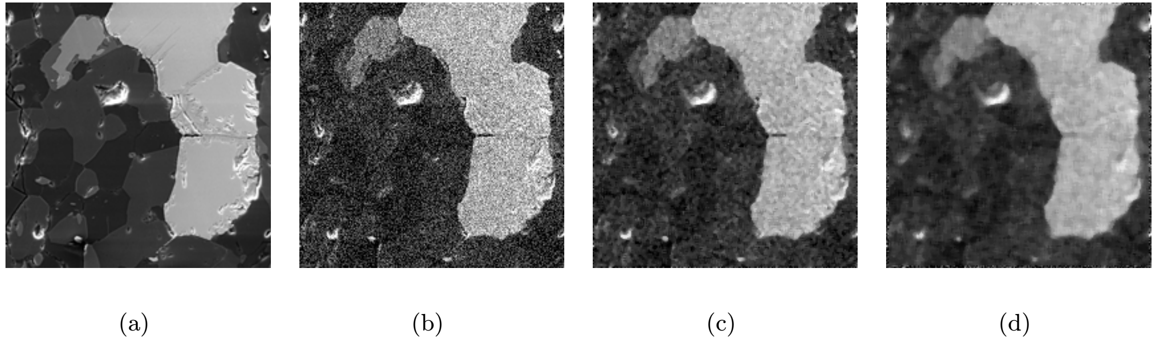
pixelů v jeho okolí. Filtry využívající tuto techniku jsou nazývány mediánové filtry.

Medián získáme tak, že seřadíme všechny pixely podle jejich hodnoty a vybereme prvek, který se nachází uprostřed. V případě, že počítáme medián ze sudého počtu prvků, postupuje se tak, že vybereme dva prvky uprostřed a bereme jejich průměr.

Mediánové filtry pracují velmi efektivně v případě odstraňování šumu typu sůl a pepř nebo impulsního šumu. To je způsobeno tím, že tento typ šumu obsahuje hodnoty, které nejsou závislé na typických hodnotách vyskytujících se v okolí. Pro gausovský šum mají podobné vlastnosti jako lokální průměrovací filtry, mnohdy však nedochází k tak výraznému rozmazání.

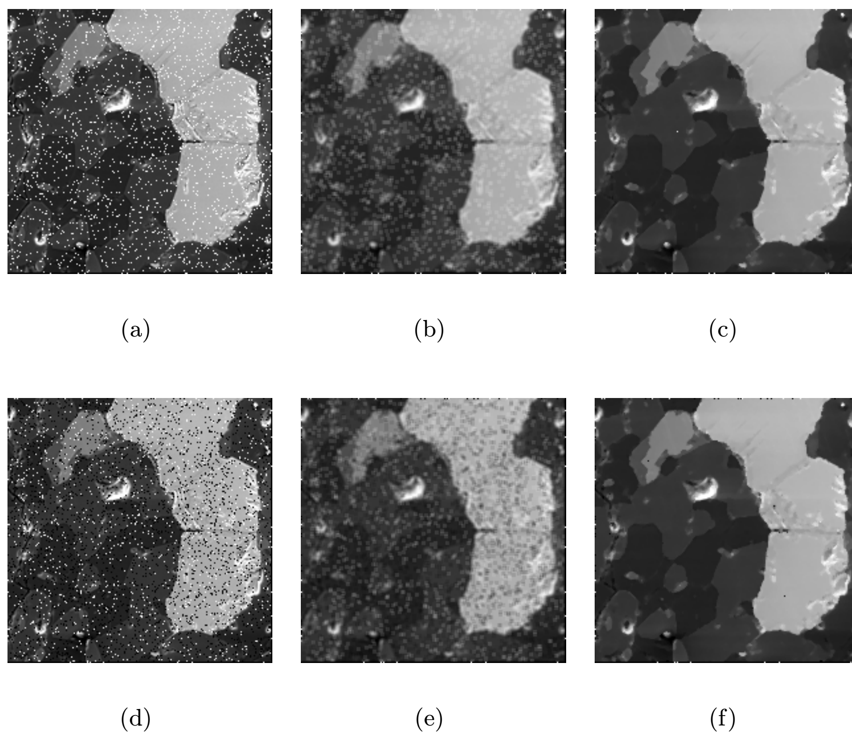


Obrázek 19: Příklad ilustrující median filtr požívající okolí 3x3



(a) (b) (c) (d)

Obrázek 20: Ukázka filtrace obrazu znehodnoceného gausovským šumem. Obraz a) originální b) poškozený gausovským šumem c) filtrovaný maskou 3x3 d) filtrovaný maskou 5x5



(a) (b) (c)

(d) (e) (f)

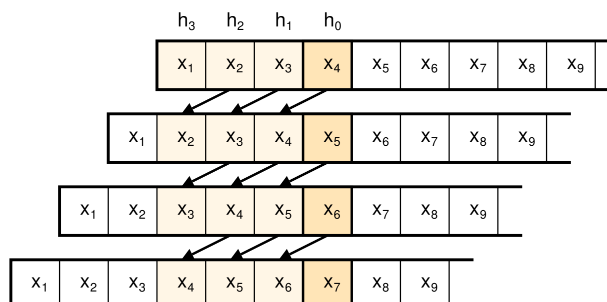
Obrázek 21: Srovnání mediánového filtru a průměrovacího filtru pro diskrétní šum. Obraz a) poškozený impulsním šumem, d) poškozený šumem typu sůl a pepř, b) a e) filtrovaný mean filtrem, c) a f) filtrovaný median filtrem 3×3

7 Obvodová realizace číslicových filtrů

Jelikož cílem této práce bude realizovat obrazový filtr netradičně pomocí vyvíjejícího se obvodu, zmiňuje se tato kapitola o několika z běžně používaných obvodových realizací číslicových filtrů. Cílem je získat alespoň základní povědomí o hardwarové realizaci filtrů, která se v praxi používá. Protože je tato oblast velmi rozsáhlá, zaměříme se pouze na realizace lineárních filtrů.

Podíváme-li se pozorněji na konvoluční rovnici 6.1.3, která popisuje obecný rekurzivní filtr IIR a na konvoluční rovnici rovnici 6.2.5, která popisuje dvourozměrný FIR filtr, nalezneme v těchto předpisech pouze dvě matematické operace. První z operací je násobení, kde jedna hodnota pochází z filtrovaného signálu a druhá (konstantní) hodnota z řady koeficientů filtru. Druhou operací je součet získaných součinů přes celé tzv. filtrovací okno. Filtrovací okno je v našem případě buď jednorozměrné nebo dvourozměrné. Ze znalosti předchozí kapitoly by nyní mělo být zřejmé, že jsme schopni realizovat libovolný lineární filtr pouze pomocí dvou matematických operací.

Filtr máme možnost implementovat v zásadě dvěma způsoby. V prvním případě můžeme výpočet chápat jako posun filtrovacího okna po signálu. Tento způsob výpočtu se ale běžně nepoužívá. Pro každý vzorek výstupu bychom museli načíst n vzorků vstupního signálu, kde n je rovno počtu prvků filtrovacího okna. Navíc tento přístup není přirozený při online zpracování signálu, kdy signál filtrem prochází. Posun filtrovacího okna lze realizovat i opačně. Filtrovací okno necháme jakoby na jednom místě, budeme však posouvat zpracovávaný signál, jak znázorňuje obrázek 22. Výsledný efekt bude stejný jako v předchozím případě, neboť každý prvek vstupního signálu se posune postupně pod všechny prvky filtrovacího okna. Můžeme si všimnout, že s každým novým výstupem stačí získat pouze jednu novou hodnotu signálu, ostatní hodnoty se jen posunou o jedno místo doleva. Schéma uvedené na obrázku 12 přesně odpovídá této metodě. Posun o jedno místo vlevo je realizován pomocí tzv. zpožďovací linky.



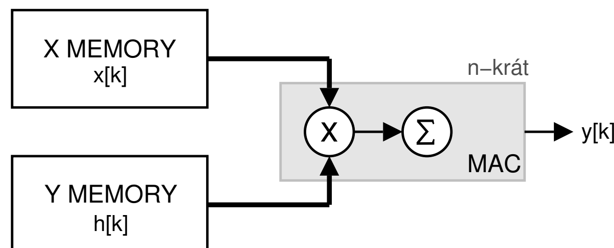
Obrázek 22: Filtrace - posun signálu pod filtrovacím oknem. Výstupní hodnota je rovna sumě součinů hodnot prvků filtrovacího okna s prvky signálu, které leží nad sebou, např. $y_4 = x_4h_0 + x_3h_1 + x_2h_2 + x_1h_3$

7.1 Řešení vycházející z konvoluce

7.1.1 DSP procesory

Ve většině případů se pro zpracování signálu používají svou architekturou speciální procesory DSP (Digital Signal Processor).Tyto procesory jsou navrženy tak, aby dokázaly tento typ aplikací zpracovávat mnohem efektivněji než běžné mikroprocesory. Jedná se o poměrně obecnou platformu, procesory je možné využít nejen v oblasti lineárního případně nelineárního zpracování signálu ale i pro řídící účely. V zásadě rozlišujeme procesory operující nad celými čísly a procesory pracující s čísly desetinnými ať už v pevné či plovoucí řádové čárce, nutno dodat, že jsou však mnohonásobně dražší. Pro většinu aplikací postačí DSP procesory, které pracují s celými čísly, jen některé specifické aplikace vyžadují zpracování v desetinné čárce. Jednou z oblastí, kde je zapotřebí preciznější výpočet je např. komunikační technologie nebo zpracování dat v medicíně.

Zatím jsme se nezmínili v čem jsou vlastně tyto procesory lepší než běžně dostupné mikroprocesory. Základní dvě operace, které se při realizaci filtrů používají, jsou násobení a sčítání, ty jsou však implementovány i v běžných mikroprocesorech. Zrychlení ale nespočívá pouze v rychlejší realizaci instrukce násobení nebo instrukce sčítání. Základním krokem konvoluce je násobení následované přičtením získané hodnoty k průběžné sumě. Kdybychom tuto operaci realizovali na běžném mikroprocesoru, bude zapotřebí k dokončení operace několik taktů. Oproti tomu na DSP procesoru násobení následované sčítáním – tzv. MAC (Multiply And Accumulate) operace trvá typicky jeden takt.



Obrázek 23: Obvodová realizace konvoluce uvnitř DSP procesoru. Pro získání hodnoty výstupního vzorku $y[k]$ je zapotřebí provést n operací MAC současně s načtením hodnot z paměti koeficientů Y a paměti vzorků signálu X.

Další výkonnostní bariérou je propustnost paměti. Je nutné si uvědomit, že při násobení musíme mít v registru jednak hodnotu koeficientu a jednak hodnotu zpožděného vzorku. Na běžném procesoru, který má klasickou harvardskou koncepci bychom na načtení obou hodnot potřebovali minimálně dva takty. DSP procesory zpravidla obsahují dva nezávislé datové proudy a DMA kanály, které se starají o přísun dat. Operaci načtení obou hodnot tedy opět zvládneme v jednom taktu. Ukázka principu je na obrázku 23.

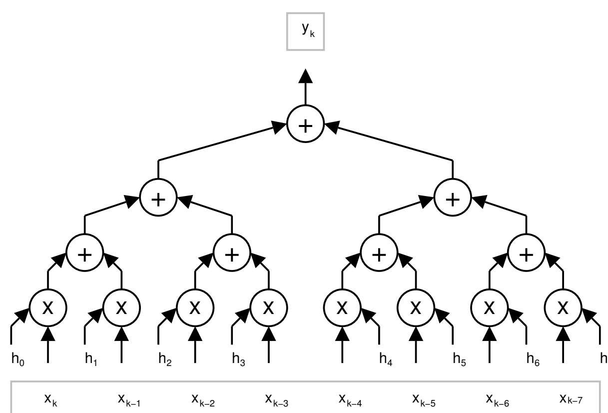
Dalším typickým rysem DSP je řada paralelních jednotek, které mohou pracovat současně. Můžeme současně načítat data ze dvou datových oblastí a současně provádět operaci MAC, čímž zredukujeme dobu potřebnou pro jeden celý krok konvoluce na jeden takt. Aby nebylo nutné při realizaci zpožďovací linky posouvat jednotlivé hodnoty v paměti, obsahují procesory

speciální adresovací modulo režim. S využitím vysokého stupně zřetězení a paralelismu se u DSP dosahuje v oblasti zpracování signálů mnohonásobně vyšší výkonnosti než u běžných procesorů.

7.1.2 HW akcelerace

V některých situacích může být sekvenční zpracování na DSP procesoru z hlediska požadované rychlosti filtru nevyhovující, neboť se vzrůstajícím počtem koeficientů roste i doba výpočtu jednoho výstupního vzorku. V praxi se běžně můžeme setkat s filtry, které jsou implementovány přímo v hardware. Tyto filtry jsou určeny především pro velmi náročné aplikace co se týče propustnosti dat. Nevýhodou tohoto přístupu hlavně u komerčně dostupných obvodů může být počet koeficientů, který je většinou limitován a tím dána i maximální přesnost či spíše kvalita filtru.

V dnešní době existuje na trhu několik rekonfigurovatelných architektur typu FPGA, které lze velmi snadno a s výhodou využít k hardwarové akceleraci filtrace. V případě využití některého z jazyků pro popis hardware (VHDL, Handel C, apod.) je díky vysoké úrovni abstrakce návrh filtrů pohodlný a jednoduchý. Tento přístup je obdobou klasického vývoje software a z něj plyne i několik positivních vlastností. Jednou z nich je, že mohou vznikat kusy kódů případně obecné generické šablony, které lze znovupoužít v jiné aplikaci. V dnešní době máme možnost např. pomocí Matlabu navrhnout výpočetní algoritmus, který se pomocí nástroje přetransformuje do VHDL [32]. Tento způsob vede na velmi výrazné zkrácení doby potřebné pro vývoj zařízení a tím i celkové ceny zařízení. Jiným typem může být aplikace, která obsahuje sadu šablon pro různé druhy filtrů, našim úkolem je pouze dosadit konkrétní parametry a nechat si vygenerovat soubor VHDL, který lze nahrát do FPGA obvodu.



Obrázek 24: Akcelerace výpočtu konvoluce pomocí stromové sčítací struktury. Hodnota y_k se získá za čtyři kroky

Abychom však nezůstali pouze v teoretické rovině, ukážeme jeden způsob zrychlení výpočtu konvoluce, který je možné realizovat v hardware. Problémem výpočtu konvoluce pomocí procesoru je sekvenční zpracování vnější sumy. Prvního zrychlení dosáhneme tak, že provedeme veškeré operace násobení paralelně. Tento krok lze realizovat v konstantním čase. Nyní

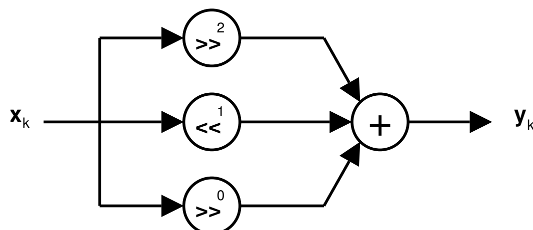
je zapotřebí provést akumulaci přes všechny dílčí součiny. Při sekvenčním zpracování se jedná o problém s lineární složitostí. Existuje však řešení, kterým lze i tuto část urychlit. Protože operace sčítání je asociativní tzn. nezáleží na pořadí sčítání $a + (b + c) = (a + b) + c$, můžeme součet realizovat postupně. Cílem je provést co nejvíce výpočtů paralelně. Z tohoto hlediska se nejlépe jeví sčítání na stromové architektuře. Architektura filtru je znázorněna na obrázku 24. Z lineární závislosti doby trvání výpočtu konvoluce na počtu koeficientů jsme přešli na závislost logaritmickou, což je celkem dramatické zrychlení.

Jako příklad uveďme 4096 koeficientů, jejichž výpočet trvá 12+1 kroků což je asi 340-ti násobné zrychlení oproti sekvenčnímu zpracování na zřetězeném DSP. Navíc můžeme jednotlivé výpočty konvoluce zřetězit, neboť procesory, které již hodnotu vypočítaly, se dále na výsledku nepodílejí. Tím získáme propustnost jeden výstupní vzorek za jeden krok, což je pro 4096 koeficientů zhruba 4096-násobné zrychlení.

7.2 Specifický přístup

7.2.1 Filtry bez násobiček

Pokud bychom realizovali lineární filtr v hardware, dojdeme k závěru, že úzkým hrdlem celého systému je výkonnost násobiček, neboť ty se z velké části podílejí na hodnotě maximálního taktovacího kmitočtu. Mimo to je násobička velmi drahý obvod z hlediska plochy na čipu, příkonu a zpoždění. Z těchto důvodů je snaha konstruovat takové filtry, které by pro svoji činnost násobičky nepotřebovaly. V případě, že používáme pevné koeficienty, lze násobičky zredukovat na jednodušší obvody. Tento přístup nás však zajímat nebude, neboť se jedná pouze o určitý způsob optimalizace. V následujícím odstavci si velmi zhruba přiblížíme metodu [1], která nám umožňuje vytvářet obecné FIR filtry.



Obrázek 25: Realizace násobení $y_k = (0.25 + 2 + 1)x_k = 3.25\ x_k$ pomocí logických posuvů

Jestliže hodnota koeficientu je rovna mocnině dvou případně součtu několika čísel s hodnotami mocniny dvou, může být násobička nahrazena součtem několika bitových posuvů vstupní hodnoty, jak ukazuje následující rovnice

$$x \cdot (2^3 + 2^1 + 2^0) = 2^3 x + 2^1 x + 2^0 x = x \ll 3 + x \ll 1 + x \ll 0 \qquad (7.2.9)$$

kde $\ll$ je operace bitového posuvu o n bitů vlevo. Bitový posuv je velmi jednoduchý realizovat v hardware, neboť se v podstatě jedná o pouhou změnu významu jednotlivých vodičů. Tímto způsobem realizované filtry dovolují vysoký taktovací kmitočet, který je limitován pouze

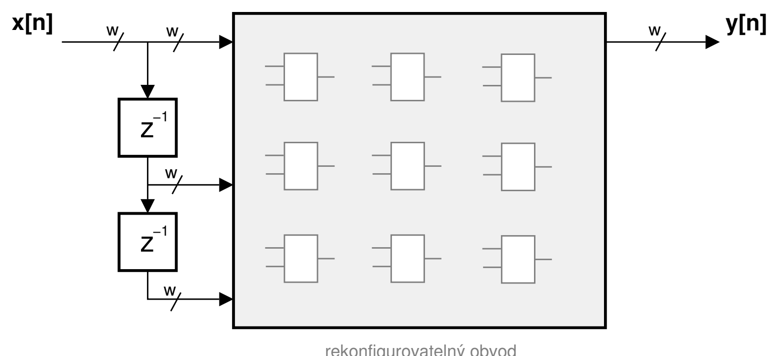
zpožděním jedné úplné sčítačky, která musí sečíst dílčí posuvy. Získáme sice vyšší rychlost a ušetříme značnou plochu čipu, ale za cenu zhoršení frekvenční charakteristiky filtru. Kvalita závisí na počtu složek mocnin dvou, které slouží jako aproximace hodnot koeficientů. Abychom dosáhli požadovaného efektu, je nutné počet složek co nejvíce redukovat. Pokud v extrému ponecháme na n bitech všech n složek, dostaneme standartní rozklad násobení a veškerá výkonnost se degraduje, neboť sčítačka musí pracovat na $2n$ bitech a sečíst celkem n hodnot.

7.2.2 Filtrace na úrovni hradel

Trendem je navrhovat obvody, které jsou velmi výkonné co se týče rychlosti a přitom zaujímají velmi malou část čipu, což je podmínka nízkého příkonu. Tyto požadavky motivují ke snaze o nalezení nových a efektivnějších hardwarových číslicových filtrů případně o nalezení nových metod filtrace.

Podobně jako do mnoha jiných i do oblasti číslicové filtrace zasáhly evoluční algoritmy. Jednou z oblastí, kde se evoluční algoritmy využívají, je hledání hodnot koeficientů filtru. Tato technika je pro nás nezajímavá, neboť nevede na žádné inovativní řešení.

Před několika lety, když se hledaly aplikace vhodné pro evoluční design, přišli někteří s myšlenkou zkusit pomocí evoluce navrhovat i IIR filtry. Evolucí na funkční úrovni, kde se jako stavební prvky použili sčítačky a násobičky, se zabývá článek [2].



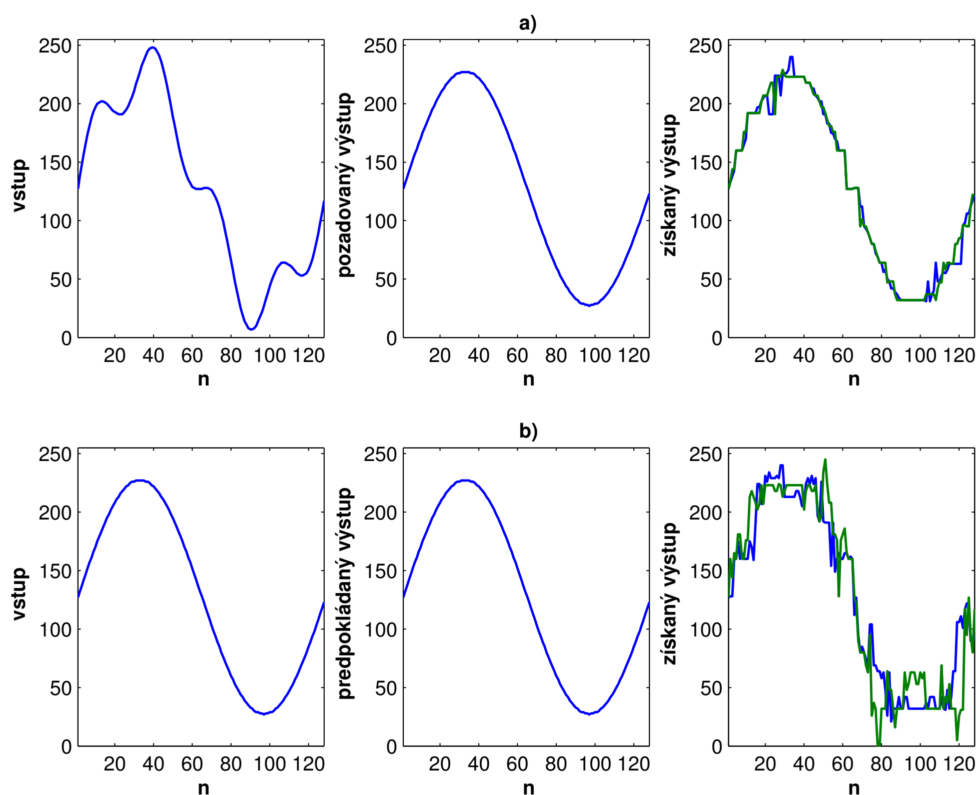
Obrázek 26: Schéma CGP pro návrh FIR filtru

Pravděpodobně nejradikálnější myšlenkou však byla snaha navrhnout číslicový filtr, který je složen pouze ze základních hradel [12, 13]. Tento přístup kompletně ignoruje techniku založenou na Multiply And Accumulate struktuře, kterou jsme se zabývali dříve. Jelikož neexistuje konvenční metoda návrhu takového filtru, bylo pro návrh filtru využito kartézské genetické programování zmiňované v kapitole 2.5. Na vstupu rekonfigurovatelného obvodu byla simulována zpožďovací linka a úkolem evoluce bylo sestavit několik hradel dohromady tak, aby realizovaly jednoduchou filtraci nad osmibitovými vzorky. Schéma je na obrázku 26, $w = 8$. Výsledky prezentované v těchto článcích byly optimistické. Podrobnějším studiem se však zjistilo, že tento přístup je bohužel velmi problematický [19]. Autor zapomněl na základní vlastnost FIR filtrů a sice linearitu. Přesto použil ve fitness funkci Fourierovu transformaci, která měla dát obraz o chování filtru přes celé spektrum. Protože základními stavebními prvky filtru jsou nelineární funkce, je velmi pravděpodobné, že systém bude také vykazovat nelineární chování, což se

potvrdilo. Ukazuje se, že pokud není garantována evolučním algoritmem linearita, evoluční návrh číslicových filtrů na úrovni hradel neprodukuje filtry, které by byly v praxi užitečné.

Získané výsledky jsou sice velmi zajímavé, neboť je možné z několika málo hradel vytvořit určitý druh filtru, který je schopen reagovat na několik vstupních signálů, rozhodně však nemůže konkurovat klasické metodě filtrace.

Na obrázku 27 je ukázka odezvy dolní propusti složené ze 75 hradel [29]. Je možné si všimnout, že pro trénovací signál je odezva filtru celkem slušná, avšak pro signál, na který nebyl filtr trénován jsou výsledky horší. Přitom v druhém případě je vstupní signál jednodušší, neboť neobsahuje rušivou vysokofrekvenční složku.



Obrázek 27: Dolní propust, průběh vstupního signálu (vlevo), požadovaného výstupního signálu (uprostřed) a signálu získaného filtrací (vpravo) pro a) trénovaný signál b) netrénovaný signál

8 Architektura PowerPC

Historie procesorů PowerPC sahá na začátek 90. let. V té době se firma IBM, společně s několika dalšími firmami, rozhodla vytvořit architekturu, která bude schopna konkurovat již značně rozšířené architektuře x86 (běžně označované jako architektura Intel). Narozdíl od velmi komplikované CISC (Complex Instruction Set) architektury však sáhla po architektuře RISC (Reduced Instruction Set), která se dnes z hlediska dalšího zvyšování výkonnosti jeví jako jedna z možných cest. Existují sice hybridní procesory x86, které mají RISCové jádro, nad kterým se provádí HW překlad CISC instrukcí, avšak v porovnání s klasickou RISC architekturou má tento přístup výkonnostní rezervu. Během HW překladu není možné provádět optimalizace a získat binární kód na takové úrovni, na jaké by jej byl schopný generovat kompilátor.

Pro architekturu RISC je charakteristická pevná délka instrukce, specializované instrukce typu L/S (Load/Store), které jako jediné mohou přistupovat k paměti a velký počet registrů pro všeobecné použití. Dalším typickým znakem je řetězená linka, zpracování instrukcí mimo pořadí a možnost vydávat několik instrukcí v jednom hodinovém cyklu. Oproti architektuře CISC dosahují procesory RISC mnohem nižšího příkonu. Na spotřebu má totiž vliv především složitost HW a délka řetězené linky, která je až čtvrtinová.

Díky značné rozmanitosti rodiny PowerPC je možné tyto procesory použít v široké škále aplikací – od vestavěných systémů přes běžné mikroprocesory až po velmi výkonné procesory určené k nasazení v serverech. Kompletní přehled můžeme nalézt např. v [36]. Nejběžněji se s PowerPC však můžeme setkat v řídicích a vestavěných systémech. Strategií IBM je dodávat procesory, které jsou co nejvíce přizpůsobeny cílové aplikaci. V současné době jsou k dispozici následující rodiny procesorů PowerPC:

Rodina 9xx je určena pro velmi výkonné aplikace. Je založena na 64-bitové architektuře, která je kompatibilní s 32bitovými aplikacemi. Aby bylo možné dosáhnout maximálního výkonu, obsahují procesory vysoce výkonné sběrnice.

Rodina 7xx nabízí 32bitové procesory, které jsou optimalizovány na výkon i spotřebu. Tyto procesory jsou určeny ke zpracování obrazu, řízení a vestavěné systémy.

Rodina 4xx zahrnuje licencovatelná 32-bitová procesorová jádra, která jsou určena pro vestavěné aplikace a systémy na čipu.

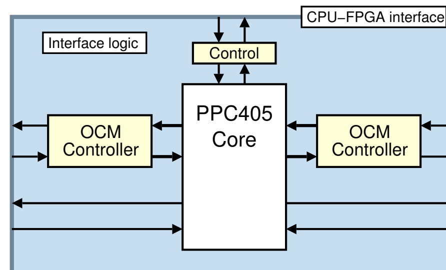
Protože z našeho pohledu je zajímavá pouze rodina 4xx, budeme se jí jako jedinou zabývat podrobněji. Rodina procesorů 4xx obsahuje dva typy procesorových jader s označením 405 a 440. Jedná se o deriváty základní architektury PowerPC. Oba typy je možné licencovat a zabudovat do zákaznického systému (na fyzické úrovni), čímž získáme systém na jednom čipu, který obsahuje relativně výkonný procesor. Procesorové jádro 405 je oproti jádru 440 mnohem jednodušší. Maximální taktovací kmitočet je 400 MHz, neobsahuje jednotku pro práci s čísly v plovoucí řádové čárce, instrukce zpracovává v pořadí a vydává maximálně jednu za takt. Jádro 440 umožňuje pracovat až na kmitočtu 660 MHz, instrukce zpracovává mimo pořadí a umožňuje vydávat dvě instrukce paralelně. Oba typy procesorů jsou vybaveny interní 16kB datovou a instrukční cache L1, PowerPC 440 má navíc podporu pro L2 cache.

8.1 PowerPC 405

Jak jsme uvedli dříve, procesor PowerPC 405 je 32-bitovou variantou architektury určené pro vestavěné systémy. V kontextu FPGA však slovo procesor znamená něco mírně odlišného a je proto nutné rozlišovat mezi procesorem a procesorovým jádrem PowerPC 405 [40]. V dalším textu budeme slovem procesor označovat celý tzv. procesorový blok integrovaný v FPGA.

Procesorový blok uvnitř FPGA Virtex II Pro tvoří následující čtyři komponenty:

- procesorové jádro IBM PowerPC 405-D5,
- dva řadiče paměti *OCM*,
- řídicí logiku a logiku hodin,
- rozhraní mezi procesorovým jádrem a FPGA.



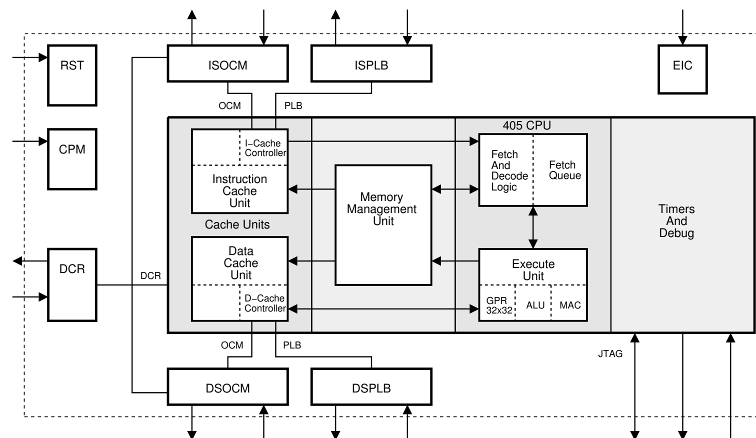
Obrázek 28: Procesorový blok

Abychom získali představu o vlastnostech a výkonnosti jádra PPC405, uvedeme si alespoň několik základních parametrů. Jádro obsahuje sadu třicetidvou 32-bitových registrů pro všeobecné použití a řadu speciálních registrů. Srdce jádra tvoří pětistupňová zřetězená linka. Většina poskytovaných instrukcí končí během jednoho taktu. Instrukční sada obsahuje instrukci MAC, pomocí které je možné efektivně realizovat řadu výpočtů (např. filtrace). Aritmetickologickou jednotku tvoří výkonná násobička s pracovním cyklem 4 takty a dělička s cyklem 35 taktů. Uvnitř jádra je integrována neblokující instrukční a datová cache, každá s kapacitou 16kB a organizací 32 bytů na řádek. Datová cache podporuje obě běžně používané strategie aktualizace paměti, tzn. write-throught (průpis) i write-back (zápis až v posledním možném okamžiku). Procesor má poměrně propracovanou podporu správy paměti založenou na stránkování, která umožňuje mapování 4GB logického adresového prostoru do fyzického.

Ze strany jádra jsou podporovány dva režimy činnosti. Privilegovaný režim, ve kterém má běžící aplikace přístup ke všem registrům a může vykonávat všechny instrukce, a restriktivnější uživatelský režim, ve kterém většinou běží uživatelská aplikace. V jádru je integrována 64-bitová časová základna a tři časovače - časovač s programovatelným intervalem (PIT), pevným intervalem (FIT) a watchdog (WDT), které mohou být inkrementovány buď s frekvencí jádra nebo samostatným hodinovým signálem.

Struktura procesorového bloku je znázorněna na obrázku 29. Jednotlivé části je možné rozdělit na interní a externí rozhraní. Interní rozhraní jsou ta, která poskytuje procesorové jádro, externí poskytuje logika procesorového bloku. Prozatím si uvedeme pouze seznam nejdůležitějších rozhraní, na která se budeme v dalším textu odkazovat.

- **C405** Processor Core – interní rozhraní,
- **CPM** Clock&Power Management – externí, řízení hodinových signálů,
- **RST** Reset Interface – externí, inicializace,
- **EIC** External Interrupt Controller – externí, řadič přerušení,
- **PLB** Processor Local Bus – externí, dedikovaná datová a instrukční sběrnice,
- **OCM** OnChip Memory Controller – externí i interní, řadič datové a instrukční paměti,
- **DCR** Device Control Register – externí i interní rozhraní.



Obrázek 29: Blokový diagram procesorového bloku Virtex II Pro

V následujících odstavcích se budeme postupně věnovat jednotlivým vstupně-výstupním rozhraním, které nabízí procesorový blok. Protože je tato problematika velmi rozsáhlá, zaměříme se pouze na nejdůležitější problémy, detaily je možné nalézt v [39, 38, 40]. Značení signálů, které budeme používat, se drží uvedených publikací. Konvence je následující: označení signálu se skládá ze dvou prefixů, které určují odkud kam je signál připojen a vlastního názvu (např. CPMC405CPUCLKEN je označením signálu CPUCLKEN, který vede z bloku řízení hodin *CPM* do jádra *C405*).

8.1.1 Rozhraní řízení hodin CPM

Protože procesorové jádro umožňuje téměř všem periferiím pracovat na kmitočtu odlišném od kmitočtu jádra, obsahuje toto rozhraní několik hodinových signálů, pomocí kterých máme možnost ovlivnit pracovní frekvenci jednotlivých komponent. Jedinou podmínkou jsou fázově zarovnané hodinové signály, což lze v FPGA velmi snadno implementovat pomocí DCM [40].

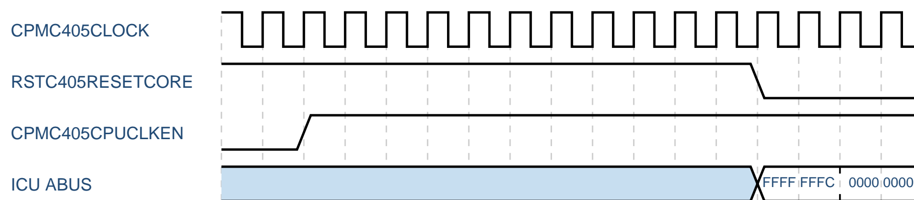
Základním hodinovým signálem je CPMC405CLOCK, který je zdrojem pro celou logiku jádra PowerPC 405 včetně časovačů. Hodinový signál je možné zakázat a povolit pomocí signálu CPMC405CPUCLKEN. Dalším velmi důležitým hodinovým signálem je signál PLBCLK, který určuje kmitočet lokální sběrnice *PLB*. Poměr mezi kmitočtem jádra a kmitočtem sběrnice *PLB* může být 1:1, 2:1, 3:1 až 16:1.

8.1.2 Rozhraní resetu RST a inicializace

Všechny nulovací signály, pomocí kterých je možné uvést periferie i procesor do počátečního stavu, jsou sdruženy do externího rozhraní *RST*. Jedná se o signály RSTC405RESETCHIP, RSTC405RESETSYS a RSTC405RESETCORE. Poslední signál slouží k hw inicializaci procesorového jádra, zbývající jsou určeny pro systémový reset periferií uvnitř procesorového bloku. Po přivedení napájecího napětí probíhá automaticky POR (power-on reset), který aktivuje všechny tři resetovací signály po dobu minimálně šestnácti hodinových taktů.

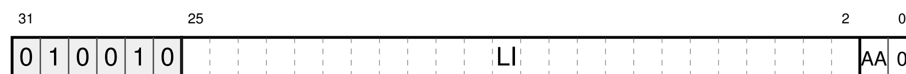
Aby proběhla inicializace procesorového jádra úspěšně, je zapotřebí splnit následující podmínky. Signál RSTC405RESETCORE musí být synchronizován (fázově zarovnaný) s hodinami jádra CPMC405CLOCK a musí být aktivní minimálně po dobu osmi hodinových taktů CPMC405CLOCK.

Inicializaci procesorového jádra můžeme provést následovně. Současně aktivujeme signál RSTC405RESETCORE a signál CPMC405CPUCLKEN, kterým povolíme činnost jádra. Minimálně osm taktů ponecháme signál RSTC405RESETCORE aktivní a poté jej deaktivujeme, čímž se procesor uvede do výchozího stavu (obr. 30).



Obrázek 30: Inicializace procesorového jádra

Bezprostředně po hw inicilizaci procesor začíná provádění kódu načtením instrukce umístěné na adrese 0xFFFF FFFC. Pokud se na této adrese nachází instrukce, která není skoková, programový čítač automaticky přeteče a procesor pokračuje načtením instrukce z adresy 0x0000 0000. Protože na adresu 0xFFFF FFFC nelze umístit více než jednu 32-bitovou instrukci, můžeme použít pouze jednoduchou instrukci větvení. Avšak i tak máme možnost adresovat paměť v rozsahu $\pm\ 2^{25}$ ($\pm$ 32MB). Formát instrukce je následující:

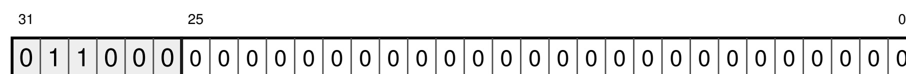


Adresa následující instrukce (NIA) se v našem případě určí pomocí následujícího předpisu:

$$\text{NIA} = (\text{AA} == 1)\ ?\ \text{EXTS}(\text{LI} \ll 2) : \text{0xFFFF FFFC} + \text{EXTS}(\text{LI} \ll 2)$$

Vezmeme-li v úvahu způsob výpočtu adresy následující instrukce, pak máme možnost se na adresu 0xFF00 0000 dostat dvěma způsoby. Buď relativním skokem zpět o 0x0FF FFFC nebo přímým skokem na adresu 0x3FF 0000. Druhá varianta využívá toho, že adresa je znaménkově rozšířena (EXTS).

Umístíme-li program od adresy 0x0000 0000 výše, je situace mnohem jednodušší. Nejčistším řešením je použít instrukci **nop** (*no operation*). Tato instrukce má následující formát:

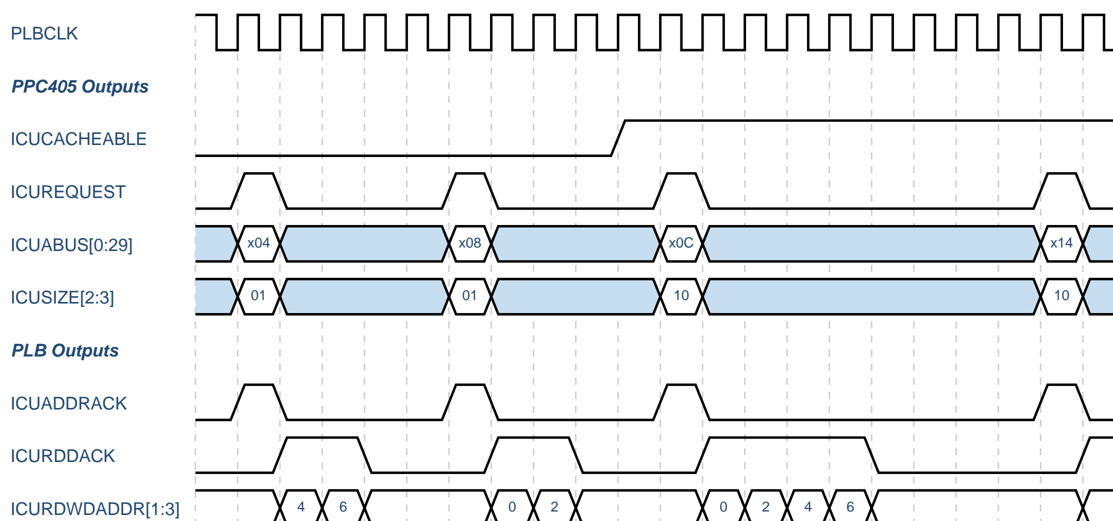


8.1.3 Sběrnice PLB

Sběrnice *PLB* je komunikačně nejsložitějším a nejvyspělejším rozhraním procesorového jádra. Jedná se o sběrnici dovolující řetězení a tím zpracování několika transakcí současně s cílem

co nejvíce překrýt komunikační režii. Aby se dosáhlo vyšší výkonnosti, obsahuje procesorové jádro dvě nezávislé *PLB* sběrnice – datovou *DSPLB* (Data-Side PLB) napojenou na řadič datové cache (DCU) a instrukční *ISPLB* (Instruction-Side PLB) sběrnici připojenou k řadiči instrukční cache (ICU). Protože *PLB* vedoucí z procesorového jádra je řízena řadiči cache a umožňuje připojit libovolný počet periferií, označuje se jako PLB master. Na úrovni FPGA čipu lze obě dedikované *PLB* sloučit a vytvořit sběrnici sdílenou. K tomu je však zapotřebí PLB arbitr.

ISPLB dovoluje ICU (Instruction Cache Unit) číst instrukce z jakéhokoliv paměťového zařízení, které je k této sběrnici připojeno. Neumožňuje však jejich zápis. Zápis do instrukční paměti je nutné řešit sdílenou sběrnicí pomocí *DSPLB*. Instrukční *PLB* má 30 bitovou adresní sběrnici a 64 bitovou datovou sběrnici. Rozhraní je navrženo tak, aby mohlo být připojeno jako *PLB* master k 64 bitové nebo 32 bitové sběrnici *PLB*. Sběrnice je schopna dosáhnout propustnosti 64/32 bitů za takt PLBCLK.



Obrázek 31: Ukázka komunikace ICU s paměťovým zařízením připojeným na sběrnici *ISPLB* (bez řetězení adresy). První dvě transakce jsou čtení slova, ostatní čtení řádky cache (čtení čtyřslova)

Oproti datové *PLB* je komunikační protokol mnohem jednodušší, neboť není nutné řešit zápis. PLB master vystaví na adresní část sběrnice ICUABUS adresu požadované instrukce a nastaví signál ICUREQ. PLB slave musí na výzvu reagovat buď chybou nebo potvrzením žádosti nastavením signálu ADDRACK. Následně je po datové části sběrnice přeneseno buď jedno 32/64 bitové slovo nebo postupně celá řádka cache čítající 8 slov (4/8 přenosů). Počet přenášených slov závisí jednak na šířce slave zařízení a jednak na tom, zda-li je povoleno využívání cache. Abychom byli schopni překrýt komunikační režii, podporuje sběrnice řetězení adresy. Bez problémů však lze na sběrnici připojit i zařízení, které řetězení nepodporuje. To je možné z toho důvodu, že pokud není zařízení schopno reagovat na další požadavek, ponechá *PLB* master požadavek aktivní tak dlouho, dokud nebude schopno zařízení reagovat. Při čtení

dat (zápis směrem do ICU) je možné data posílat v libovolném pořadí - buď poslat nejprve požadované slovo a poté zbytek nebo slova posílat sekvenčně (nejjednodušší varianta).

DSPLB dovoluje DCU (Data Cache Unit) číst a zapisovat data z jakéhokoliv paměťového zařízení připojeného na tuto sběrnici. Datová *PLB* obsahuje 32 bitovou adresní část sběrnice a dvě 64 bitové datové sběrnice, jednu pro směr z procesoru (zápis) a druhou pro směr opačný (čtení). Rozhraní je navrženo tak, aby mohlo být připojeno jako *PLB* master k 64 bitové nebo 32 bitové sběrnici *PLB*. Sběrnice je schopna dosáhnout propustnosti 64/32 bitů za takt PLBCLK.

Komunikační protokol je podobný jako u *ISPLB*. Jelikož je však podporován i zápis, je nutné rozlišovat mezi čtecí a zápisovou transakcí. K tomu účelu slouží signál DCURNW, který v aktivním stavu indikuje čtení. Protože instrukční sada podporuje zápisy s různou datovou šířkou, je nutné brát při implementaci v úvahu i signál DCUBE. Tento signál určuje, který byte, půlslovo nebo slovo se má do paměti zapsat.

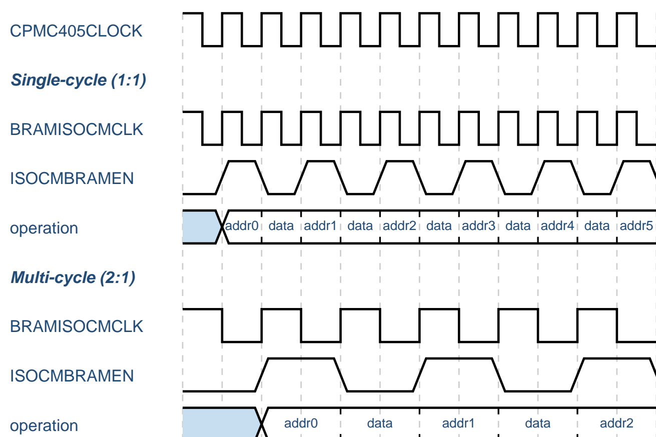
8.1.4 Rozhraní OCM

Externí řadič *OCM* poskytuje rozhraní mezi pamětmi typu block RAM distribuovanými uvnitř FPGA a signály interního *OCM* rozhraní procesorového jádra. Podobně jako *PLB* je i *OCM* tvořeno nezávislou instrukční a datovou větví. Rozhraní je navrženo tak, aby množství logiky mezi rozhraním a pamětí bylo co nejmenší. K realizaci instrukční nebo datové paměti je zapotřebí pouze kombinační logika (multiplexory a logická hradla). V některých speciálních případech (malé množství připojených block RAM) ani žádnou logiku nepotřebujeme.

Jako většina externích periferií umožňuje i řadič *OCM* pracovat na frekvenci lišící se od frekvence jádra. Na rozdíl od hodinových signálů ostatních zařízení, které jsou součástí rozhraní *CPM*, tvoří hodinové signály součást rozhraní *OCM*. Protože procesorový blok obsahuje dva fyzicky oddělené řadiče (instrukční *ISOCM* a datový *DSOCM*), lze pracovní kmitočet definovat pro každý řadič nezávisle pomocí signálu BRAMISOCMCLK a BRAMDSOCMCLK. Poměr mezi kmitočtem jádra a kmitočtem rozhraní *OCM* může být 1:1, 2:1, 3:1 nebo 4:1. Jelikož externí *OCM* nepodporuje vkládání čekacích stavů ani potvrzovací schéma, je nutné řadiči explicitně sdělit, jaký je mezi hodinovými signály poměr. Poměr určují tři nejméně významné bity signálové sběrnice DSCNTLVALUE[7:0] a ISCNTLVALUE[7:0]. Podrobnějším studiem bychom zjistili, že bity udávají počet hodinových cyklů jádra, po který má řadič čekat, než požádá o další přenos.

Maximální dosažitelný výkon instrukční *OCM* je ekvivalentní instrukční *PLB*. Jedno čtení z paměti trvá dva takty BRAMISOCMCLK – v prvním taktu je vystavena adresa, v druhém taktu se čtou data. Tím, že *ISOCM* má šířku dat 64 bitů, poskytuje přístup do instrukční paměti s propustností jedna instrukce za jeden hodinový cyklus BRAMISOCMCLK. Pokud je frekvence BRAMISOCMCLK shodná s frekvencí jádra, získáme přístup, jenž je ekvivalentní zásahu do cache. Abychom však této výkonnosti dosáhli, musí být návrhový software schopen rozmístit a propojit design tak, aby bezchybně pracoval na frekvenci jádra, což je většinou v případě použití většího množství paměti a složitějšího systému nemožné. Tím se však stává *OCM* mnohem méně výkonnějším než *PLB*, neboť nelze povolit využívání cache. To, že data neprocházejí pamětí cache, může být výhodné např. v situaci, kdy by často docházelo k trashingu (nechtěnému uvolňování a znovunačítání obsahu řádky cache).

U datové *OCM* jsme na tom s maximální výkonností poněkud hůře, neboť ta je oproti datové *PLB* poloviční. Čtení i zápis sice trvají podobně jako v případě instrukční větve dva takty BRAMDSOCMCLK, avšak šířka datové sběrnice je poloviční.



Obrázek 32: Ukázka komunikace a) v režimu 1:1 (single cycle), b) v režimu 2:1 (multi cycle)

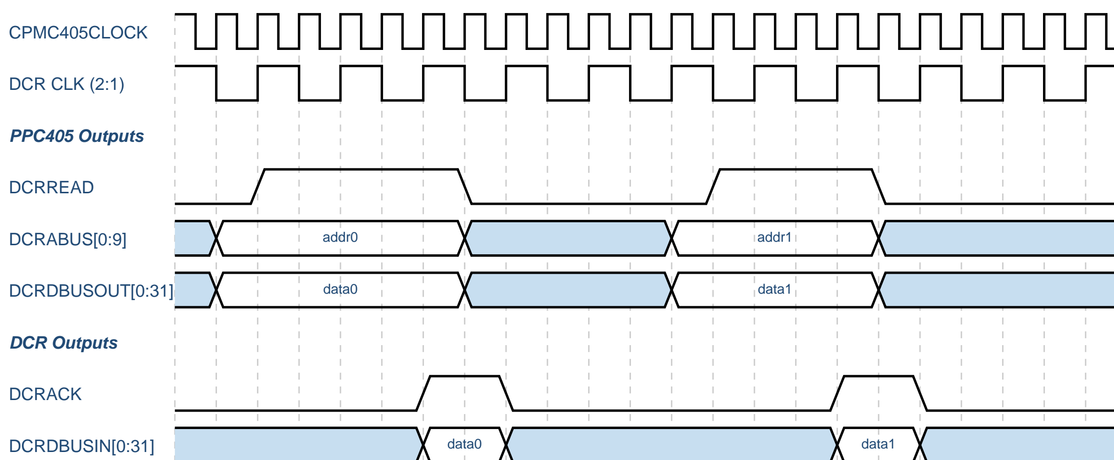
Protože rozhraní *DSOCM* je navrženo pro připojení paměti, je nutné v případě použití periferií, které nemají charakter paměti (např. do paměti mapované V/V periferie), dát pozor na skrytá úskalí. Řadič *OCM* nezaručuje zachování pořadí provádění jednotlivých operací čtení a zápisu. Může se stát, že zápis bude vykonán později než čtení i když instrukce zápisu (store) je procesorem zpracována dříve než instrukce čtení (load). Dalším problematickým místem je zápis následovaný čtením ze stejné adresy. V takovém případě může být čtení vyřízeno jako interní operace (forwarding) uvnitř řadiče, což znamená, že data načtená procesorem nejsou data vrácená periferií, ale data z interního bufferu *OCM*. Oba problémy je nutné řešit na úrovni zdrojového kódu.

8.1.5 Rozhraní DCR

Rozhraní *DCR* poskytuje prostředek, kterým dovoluje procesoru řídit a inicializovat části systému, které nejsou součástí procesorového jádra. Na interní *DCR* rozhraní jsou připojeny oba externí *OCM* řadiče (datový i instrukční), ke kterým by jinak nebylo možné z procesoru přistupovat a konfigurovat je. Jedná se vlastně o 32-bitový paralelní port, pomocí kterého je možné číst a měnit obsah registrů externích zařízení. Jednotlivá zařízení a jejich registry je možné adresovat pomocí 10-bitové adresy. Protože je *DCR* vyvedeno i z procesorového bloku (externí *DCR*), je možné je s výhodou využít k ovládání externích periferií umístěných uvnitř FPGA čipu.

Drobným nedostatkem tohoto rozhraní může být, že uvnitř FPGA Virtex II Pro běží *DCR* na kmitočtu jádra a jako jediný externí blok nemá samostatný hodinový signál. Avšak *DCR* již z principu umožňuje, aby připojená „slave“ zařízení mohla pracovat na libovolném kmitočtu (vyšším i nižším). Pro komunikaci je použito potvrzovací schéma pomocí signálu ACK (Handshake). Jedinou podmínkou, která je kladena na taktovací kmitočet je, že musí

být odvozen od stejného zdroje, jinými slovy fázově zarovnaný. Je nutné si dát pozor na to, že pokud do 64 taktů není potvrzena žádost o čtení/zápis, procesor pokračuje bez chyb další instrukcí. Maximální propustnost tohoto rozhraní je při taktovacím kmitočtu 300MHz a kombinačně generovaným potvrzovacím signálem 300 MB/s (za předpokladu, že komunikace trvá čtyři takty).



Obrázek 33: DCR handshake v režimu 2:1, DCRACK generován sekvenčně se zpožděním 2 takty

Na *DCR* je možno připojit libovolný počet zařízení. Sdílení se provede tak, že se datové vstupy a výstupy jednotlivých zařízení zapojí do řetězce. V závislosti na výstupu adresového komparátoru se pomocí multiplexoru na datový výstup přepíná buď obsah datového registru nebo datový vstup. Přístup na rozhraní ze strany procesoru je umožněn pomocí specifických instrukcí **mtdcr** (*move to dcr*) a **mfdcr** (*move from dcr*).

8.1.6 Řadič přerušení EIC a obsluha přerušení

Přerušení a výjimky jsou děleny do dvou tříd – kritické a nekritické. Každá třída má vlastní registrový pár, který slouží k uložení návratové adresy. Toto rozdělení umožňuje obsluhovat nezávisle přerušení obou tříd, tzn. kritické přerušení může přerušit obsluhu nekritického přerušení. Návratová adresa je ukládána automaticky a obnovována automaticky při návratu z přerušení pomocí instrukcí **rfi** (*return from interrupt*) a **rfci** (*return from critical interrupt*).

Kromě řady interních přerušení, které jsou pro nás nyní nepodstatné, existují i oba typy externích přerušení. Pro žádost o kritické přerušení slouží signál EICC405CRITINPUTIRQ a pro nekritické signál EICC405EXTINPUTIRQ. Protože obsluha přerušení reaguje na hladinu a nikoliv na hranu, je zapotřebí před návratem z přerušení žádost o přerušení stáhnout, jinak dojde k dalšímu vyvolání obslužné rutiny. Nejjednodušší variantou je využít k potvrzení žádosti o přerušení a následnému zrušení žádosti externí rozhraní *DCR*.

Obsluha přerušení se skládá z následujících kroků. Po dokončení právě rozpracované instrukce je nejprve určena třída, do které generované přerušení patří a do příslušného regis-

trovaného páru je uložena adresa následující instrukce. V dalším kroku procesor určí adresu rutiny obsluhy přerušení. Adresa je vypočtena jako součet 32-bitové báze uložené v registru EVPR (Exception Vector Prefix Register) a ofsetu, jenž je pevně přiřazen jednotlivým typům přerušení. Nekritickému externímu přerušení je přidělena hodnota 0x0500, kritickému hodnota 0x0100. Oba typy externích přerušení je možné maskovat příslušnými bity ve stavovém registru MSR. Po návratu z přerušení je z příslušného registrového páru obnovena adresa následující instrukce a program může pokračovat instrukcí, před kterou nastalo přerušení.

8.1.7 Shrnutí

Na závěr kapitoly věnující se procesoru PowerPC stručně shrňme základní parametry komunikačních rozhraní. Instrukční rozhraní *OCM* umožňuje adresovat až 16MB instrukční paměti, výkon *OCM* je srovnatelný se zásahem do cache, ovšem pouze v případě, že pracuje na frekvenci shodné s frekvencí procesorového jádra. Jelikož je v praxi velmi problematické, aby návrhový software navrhl uvnitř FPGA rozhraní pracující na tak vysoké frekvenci, dostává se do výhody komunikačně složitější instrukční *PLB*, kde je možné zapnout podporu cache. K *OCM* je problematické připojit periferie, které nemají charakter paměti, neboť je nutné softwarově vkládat prázdné takty. Navíc neumožňuje připojit periferie, které mají latenci větší než jeden takt. Datová *OCM* má oproti instrukční výkon poloviční, vyplatí se proto pouze tehdy, pokud nepožadujeme extrémně rychlé přenosy.

PLB je velmi komplexní sběrnice, umožňuje připojit libovolný typ zařízení a podporuje variabilní latenci. Zařízení připojená na sběrnici lze hierarchicky členit a vytvářet mosty mezi *PLB* a jinými typy sběrnic. Menší nevýhodou oproti *OCM* jsou vyšší nároky na hardware řadiče, který musí řídit mnohem více signálů. Na druhou stranu představuje *PLB* výkonnostní špičku.

Posledním typem rozhraní je *DCR*, které se používá zejména k zpřístupnění registrů externích zařízení. Vzhledem k složitosti řadiče a výkonnosti se jedná o optimální řešení. Nevýhodou je nemožnost přímo ovlivnit pracovní kmitočet pomocí externího hodinového signálu, avšak i přesto je možné díky handshake protokolu pracovat s téměř libovolnou latencí.

Srovnání uvedených sběrnic z hlediska několika základních parametrů je uvedeno v následující tabulce. V posledním sloupci je pro získání lepší představy uvedena výkonnost procesorového jádra.

rozhraní	**DCR**	**ISOCM**	**DSOCM**	**ISPLB**	**DSPLB**	**C405**
max. propustnost [MB/s]	300	1 200	600	2 400 [1]	2 400 [1]	1 200
šířka datové sběrnice [b]	32	64	32	64	64	32
adresovatelný prostor [B]	1 k	16 M	16 M	4 G	4 G	4 G
podpora variabilní latence	ano	ne	ne	ano	ano	n/a

[1] Maximální propustnost je vyšší, než propustnost procesorového jádra (cache), neboť operace s *PLB* se provádí přes fill buffer, trvalá propustnost se tedy pohybuje na hranici propustnosti jádra

Tabulka 1: Srovnání základních parametrů jednotlivých rozhraní

9 Rekonfigurovatelný systém

Tato kapitola se detailněji zabývá praktickou implementací rekonfigurovatelného systému využívajícího procesor PowerPC. První část je věnována možnostem realizace. Postupně jsou představeny a zhodnoceny tři různé typy rekonfigurovatelných architektur. Druhá část se zabývá praktickou implementací evoluční platformy.

Cílovou platformou, na které bude rekonfigurovatelný systém realizován, je PCI karta COMBO6X vyvinutá v rámci projektu Liberouter [37], která je primárně určena k akceleraci síťových operací (směrování, analýzu toku dat, apod.) v gigabitových sítích. Tato karta obsahuje dva FPGA obvody Virtex II Pro a umožňuje připojení rozšiřujících karet. Jednodušší FPGA obvod XC2VP4-7 (FG456) má na starosti řízení sběrnice PCI, pomocí které může komunikovat s PC. Druhý FPGA obvod Virtex II Pro XC2VP50-7 (FF1517) je určen pro implementaci vlastní aplikace a obsahuje více než 40000 registrů.

Pro komunikaci s PC je vytvořena uvnitř FPGA infrastruktura založená na sběrnici LocalBus, která dosahuje propustnosti 200 MB/s na 50MHz. Menší FPGA obvod zajišťuje transformaci interního protokolu LocalBus na PCI a naopak.

9.1 Možnosti realizace

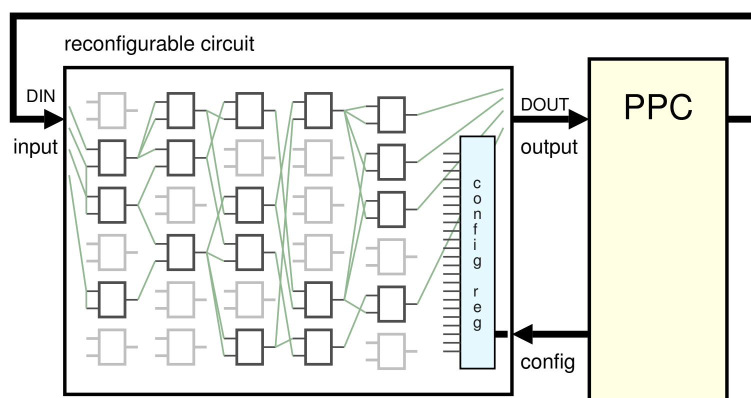
Cílem této kapitoly je ukázat několik odlišných způsobů realizace vyvíjejícího se systému, který využívá procesor PowerPC. U každé navržené architektury bude snahou alespoň naznačit jak její přednosti tak i zásadní nedostatky. Zaměříme se pouze na intrinsickou evoluci na čipu, kdy je rekonfigurovatelný obvod reprezentován pomocí *VRC*, který jsme diskutovali v kapitole 5.3.3. Předností *VRC* je nejen možnost velmi rychlé rekonfigurace, ale i libovolné přizpůsobení cílové aplikaci a to jak strukturou pole, tak funkcí jednotlivých elementů. Protože evoluční algoritmus i rekonfigurovatelný obvod jsou implementovány uvnitř FPGA, můžeme hovořit o rekonfigurovatelném systému na čipu.

Na první pohled se může zdát, že použití procesoru PowerPC pouze celou architekturu komplikuje. Evoluční algoritmus je sice možné realizovat kompletně v hardware [20], narazíme ovšem na několik problematických míst. Prvním nedostatkem je složitost genetické jednotky, kvůli které nelze pracovat na frekvenci srovnatelné s frekvencí procesorového jádra. Avšak mnohem závažnějším problémem je nutnost zjednodušit evoluční algoritmus a použité operátory tak, aby bylo možné algoritmus v hardware rozumně implementovat. Právě tento důvod vede na myšlenku přesunout výpočetně složitější operace do procesoru (software) a zbylou část (časově náročnější) řešit v hardware. Předností integrace procesoru PowerPC do hradlového pole FPGA je možnost bezproblémově připojit široké paralelní sběrnice, které pracují na relativně vysokém kmitočtu. Tím jsme schopni maximálně využít kapacitu sběrnic a dosáhnout velmi vysokého výkonu.

Abychom zjednodušili bloková schemata, budeme v následujícím textu blokem *PPC* označovat seskupení, které je tvořeno procesorovým blokem uvnitř Virtex II Pro a pro chod procesoru nezbytnými periferiemi jako jsou instrukční paměť, datová paměť, řídicí logika apod.

Nejjednodušší architektura vyvíjejícího se systému, která využívá procesor PowerPC, je uvedena na obrázku 34. Celý systém se skládá pouze z virtuálního rekonfigurovatelného obvodu *VRC*, procesorového bloku *PPC* a řídicí logiky. Procesor má možnost pomocí jedné sběrnice

obvod rekonfigurovat, pomocí druhé sběrnice nastavovat *VRC* obvodu vstupy a pomocí třetí sběrnice získat výstup. Protože tato architektura neumožňuje kvůli sekvenčnímu přístupu provádět současně rekonfiguraci a evaluaci, stačí použít pouze jednu sdílenou sběrnici (např. *PLB*). Z hlediska počtu sběrnic se jedná o prakticky realizovatelné řešení, neboť procesor jich poskytuje dostatek.



Obrázek 34: Blokový diagram rekonfigurovatelné architektury využívající PowerPC

Evoluční cyklus se skládá z následujících tří kroků. Pro každé kandidátní řešení musí procesor nejprve postupně nahrát odpovídající konfigurační řetězec do *VRC* a tím provést jeho rekonfiguraci. V dalším kroku probíhá ohodnocení kandidátního řešení. Na sběrnici D_{IN} procesor generuje všechny vektory trénovací množiny a pomocí sběrnice D_{OUT} získává reakci rekonfigurovatelného obvodu, na základě které je průběžně počítána fitness hodnota. Po získání fitness hodnoty jednoho řešení je zapotřebí znovu provést rekonfiguraci a ohodnocení. Po ohodnocení všech jedinců je zapotřebí vytvořit novou populaci a jednotlivé kroky opakovat.

Protože není nutné provádět softwarovou simulaci obvodu, dosahuje tento systém oproti softwarovému řešení vyšší výkonnosti. Tím, že rekonfigurovatelný obvod realizujeme čistě kombinační logikou, jsme schopni odezvu získat mnohem rychleji, než simulací v software. Operace nad populační pamětí mohou být také velmi rychlé, neboť vzhledem k její velikosti není problém ji celou umístit do datové paměti cache, která má 16kB.

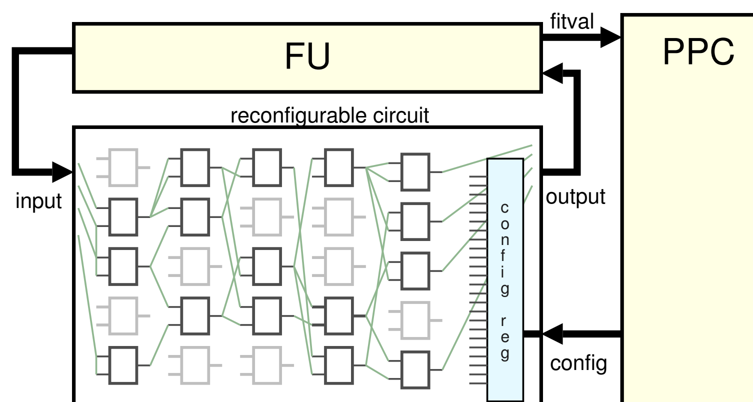
Nedostatkem této architektury je nutnost nahrávat v každém kroku celý konfigurační řetězec přes relativně úzkou sběrnici do *VRC* obvodu. Přenos konfigurace do *VRC* musí být serializován, v nejlepším případě po 64 bitech při použití sběrnice *PLB*. Z hlediska CGP a běžně používaných operátorů se ale jedná o zbytečnou režii, která v software nenastává. Tento nedostatek je možné velmi snadno odstranit zavedením paměti block RAM (BRAM) mezi *VRC* a *PPC*. Mimoto může BRAM obsahovat několik konfigurací, v ideálním případě celou populaci. Jelikož se používá pouze operátor mutace, stačí ze strany PowerPC nahrát do BRAM jen několik odlišných bytů. Tím, že je možné ze strany *VRC* vytvořit sběrnici o libovolné šířce, bude v tomto seskupení rekonfigurace *VRC* trvat pouze tolik taktů, kolik má *VRC* obvod sloupců.

I po předchozím vylepšení má však tato varianta stále výkonnostní rezervu. Procesor je sice využit naplno, ale dělá množství zbytečné práce, kterou může dělat jednoduchý hardware.

Zásadním problémem je, že většinu času procesor stráví ohodnocováním jedince a kvůli absenci DMA přenosů není možné překrýt fázi generování nové populace s rekonfigurací *VRC*. Pokud bude generování nové populace časově náročnější, ztratí se zbytečně mnoho taktů. Dramatický vliv na výkonnost může mít v případě nevhodné realizace i latence jednotlivých rozhraní procesoru. Jelikož veškerá inteligence je skryta v procesoru, může být mírnou komplikací složitost řídicího programu.

Kromě nevýhod zmiňme i některé pozitivní vlastnosti. V některých situacích je výhodné, že procesor má možnost řídit veškerou činnost a mít tím vše pod kontrolou. Největší výhodou je však jednoduchost architektury, není zapotřebí řešit přerušení a relativně složité řízení na úrovni hardware.

Jelikož cílem je dosáhnout maximálního výkonu, musíme se poohlédhout po jiné architektuře. Hlavním nedostatkem předchozí architektury je nemožnost překrýt některé části evolučního algoritmu užitečným výkonem. Abychom tento nedostatek odstranili, bude zapotřebí dovolit vyšší stupeň paralelismu. Logickým krokem je přesunout část starající se o evaluaci do hardware a tím značně odlehčit procesoru, který bude mít na starosti pouze tvorbu nové populace. Jednotlivé kroky evoluce je sice z principu evolučního algoritmu stále nutné provádět sekvenčně, avšak přínosem je to, že procesor bude méně zatížen. Tím dostane více prostoru k tomu, aby byl schopen během evaluace připravit další konfigurační řetězec a nemusel se přípravou zabývat až po skončení evaluace, jak tomu bylo v předchozí variantě.



Obrázek 35: Architektura využívající hardwarové fitness jednotky a procesoru

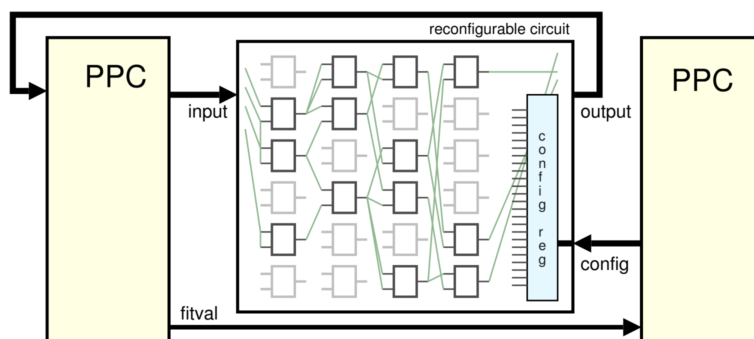
Architektura dovolující vyšší výkonnost je uvedena na obrázku 35. Celý systém se skládá z virtuálního rekonfigurovatelného obvodu *VRC*, fitness jednotky *FU*, bloku *PPC* a řídicí logiky. Procesor má možnost pomocí jedné sběrnice obvod rekonfigurovat a pomocí druhé sběrnice číst výslednou fitness hodnotu z jednotky *FU*. Mimoto jsou zapotřebí ze strany procesoru řídicí signály k ovládání fitness jednotky. Podobně jako u předchozí varianty budeme uvažovat mezi blokem *PPC* a *VRC* populační paměť BRAM přístupnou např. pomocí *DSOCM*. Řízení FU a čtení fitness hodnoty je možné např. pomocí 32-bitového rozhraní *DCR*. Z hlediska sběrnic je situace opět vyhovující, protože je sběrnic dostatek a není nutné multiplexování.

Kromě překrytí fáze evaluace a generování nové populace je možné překrýt i fázi rekonfigurace a evaluace. Pokud místo kombinačního rekonfigurovatelného obvodu *VRC* použijeme

sekvenční řetězený *VRC*, můžeme po nakonfigurování prvního sloupce *VRC* již spustit evaluaci. Pokud bude PowerPC schopen během evaluace jednoho kandidátního obvodu připravit a nahrát do BRAM konfiguraci pro další obvod, získáme systém bez jediného nevyužitého taktu. V případě evoluce složitějších (vícevstupových) obvodů není problém předchozí podmínku zaručit.

Stejně jako předchozí řešení je i tato varianta vhodná pro libovolnou aplikaci kartézského genetického programování (evoluční hledání filtrů, kombinačních obvodů, polymorfních obvodů apod.). S jiným typem řešeného problému se mění pouze složení elementů rekonfigurovatelného obvodu *VRC* (struktura zůstává, lišit se mohou použité funkce) a případně i způsob výpočtu fitness hodnoty.

Mírnou komplikací je složitost hardwarové části a nutnost implementovat na straně procesoru podporu přerušení. Sice jsme odstranili všechny nevýhody předchozí architektury a získali systém s maximální propustností, ale použitím hardwarové jednotky *FU* jsme vnesli nový nedostatek. Hardwarová fitness jednotka může realizovat pouze jednoduché operace jako je sčítání (odčítání) nebo násobení. Pokud bude zapotřebí počítat např. průměr přes všechny hodnoty, nastanou značné komplikace s implementací na úrovni hardware.



Obrázek 36: Architektura využívající oba procesory

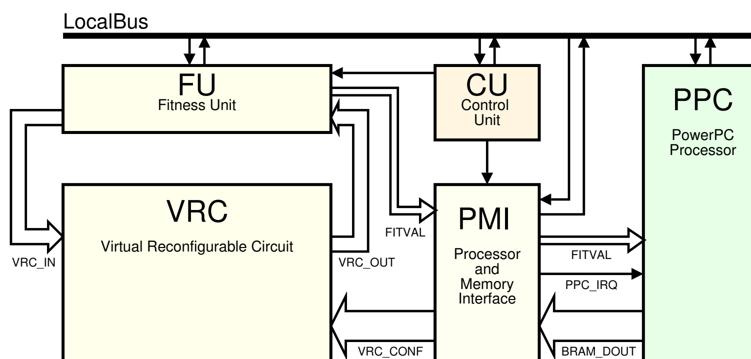
Nedostatek předchozího řešení odstraňuje architektura uvedená na obrázku 36. Jelikož FPGA Virtex II Pro obsahuje dva procesorové bloky, je možné využít jeden procesor jako fitness jednotku a druhému svěřit generování nové populace. Uvedená architektura je výkonnostně ekvivalentní s předchozí variantou, neboť propustnost i šířka sběrnic PowerPC mnohonásobně převyšuje požadavky rekonfigurovatelného obvodu *VRC*. Přesun hardwarové fitness jednotky do software umožňuje řešit i složitější výpočty fitness hodnoty.

Komplikace však přináší nutnost řešit vzájemnou synchronizaci procesorů. Pravý řídicí procesor musí nejen dávat povely levému procesoru a *VRC*, ale i číst fitness hodnotu vypočítanou levým procesorem. K zajištění synchronizace stačí mít možnost generovat na obou stranách přerušení. Problematické může být zajistit synchronizaci tak, abychom dosáhli co největšího překrytí a nulové latence. Jako nejlepší varianta se jeví nechat levý procesor zapsat fitness hodnotu do registru nebo paměti FIFO, nastavit žádost o přerušení a začít ohodnocovat další kandidátní obvod. Pravý procesor si v obsluze přerušení přečte hodnotu z registru a může začít vytvářet novou konfiguraci. Tím je latence obsluhy přerušení úplně odstraněna.

9.2 Praktická realizace

Navržený systém vychází z architektury uvedené na obrázku 35. Architektura zmíněná jako poslední sice poskytuje komfortnější řešení, avšak její implementace by se vyplatila pouze za předpokladu, že by výpočet fitness hodnoty byl v hardware obtížně realizovatelný. To však není náš případ.

Jelikož je k dispozici čistě hardwarová realizace evolučního algoritmu určená k evolučnímu návrhu obrazových filtrů, bude snahou na tuto architekturu pokud možno navázat [20]. Architekturu tvoří následující čtyři bloky: fitness jednotka, rekonfigurovatelné pole, řídicí jednotka a genetická jednotka. Úkolem bude nahradit hardwarovou genetickou jednotku procesorem PowerPC, upravit řídicí logiku a navrhnout vhodný evoluční algoritmus, který dovolí maximální překrytí jednotlivých fází výpočtu.

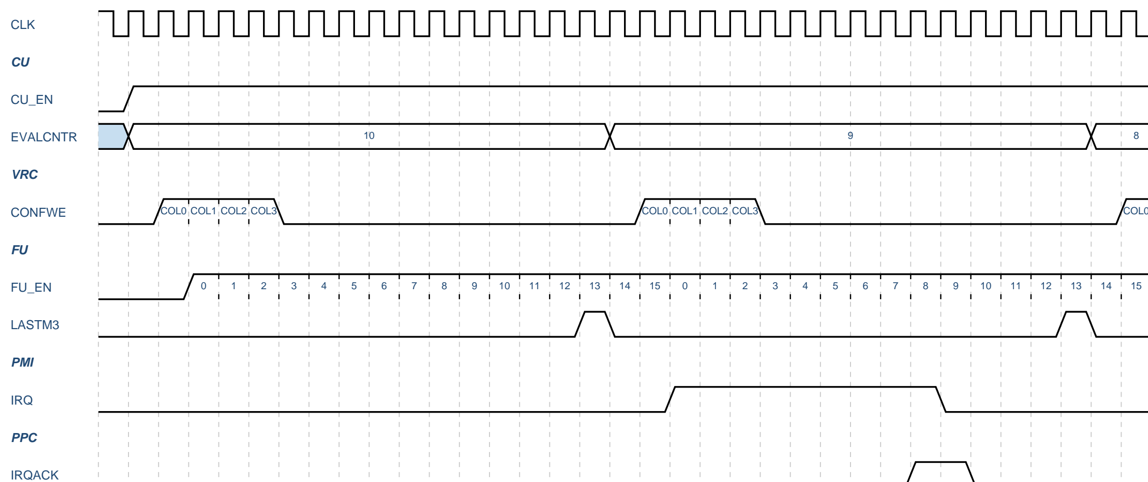


Obrázek 37: Blokový diagram rekonfigurovatelné architektury

Struktura navrženého systému je uvedena na obrázku 37. Celý systém je řízen řídicí jednotkou *CU* (Control Unit), která je připojena na sběrnici LocalBus. Pomocí této sběrnice je schopna ze strany PC přijímat příkazy. Lze zvolit maximální počet generací, po který má evoluce běžet, spustit, pozastavit nebo zastavit evoluci. Procesor PowerPC v tomto přístupu hraje roli podřízené jednotky, která musí na žádost vytvořit nový konfigurační řetězec. Realizace opačného přístupu, kdy je systém řízen procesorem, je sice možná, ale nepřináší žádné zlepšení, spíše naopak. Pokud bychom chtěli k řízení celého systému využít sběrnici LocalBus, musel by procesor ve smyčce testovat, kdy má spustit evoluci. Mnohem horší je však testovat, zda má procesor evoluci ukončit. Takový test by bylo nutné realizovat buď ve smyčce, což představuje výrazné zhoršení výkonnosti, nebo pomocí kritického přerušení.

Kromě komunikace s PC zajišťuje řídicí jednotka synchronizaci celého systému. Jeden krok evoluce se skládá z následujících podkroků. Nejprve *CU* požádá jednotku *PMI* (Processor And Memory Interface), aby zahájila nahrávání konfiguračního řetězce do *VRC* (CONFWE). Jakmile *PMI* nakonfiguruje první sloupec rekonfigurovatelného obvodu, vydá *CU* povel fitness jednotce *FU* (Fitness Unit), která započne vyhodnocování obvodu (FU_EN). Několik taktů před evaluací posledního testovacího vektoru oznámí fitness jednotka řídicí jednotce, že bude končit (LASTM3). Ta reaguje nastavením žádosti o zahájení nahrávání další konfigurace a celý cyklus se opakuje. Jakmile je obvod ohodnocen, pošle fitness jednotka výsledek jednotce *PMI*, která se postará o její propagaci do procesoru (IRQ). Jelikož je celá komunikace řetězena,

jednotlivé evaluace na sebe navazují bez vzniku nevyužitých taktů, což demonstruje diagram 38. Jediný blok, který může komunikovat s procesorem PowerPC, je *PMI*. Tento blok se stará o zaslání fitness hodnoty a rekonfiguraci rekonfigurovatelného obvodu. Komunikace s procesorem probíhá prostřednictvím přerušení nastavením signálu IRQ (žádosti o přerušení). Ukončena je potvrzením přerušení ze strany procesoru (IRQACK), který během přerušení nachystal novou konfiguraci.



Obrázek 38: Průběh prvních dvou z deseti kroků evoluce. *VRC* tvoří čtyři sloupce a obsahuje čtyři vstupy, konfigurace tedy trvá čtyři takty a ohodnocení $2^4 = 16$ taktů

Nyní zbývá pouze dořešit: 1. jakým způsobem se budou přenášet data mezi procesorem a rekonfigurovatelným obvodem VRC, 2. jakou použít sběrnici, abychom dosáhli co nejvyššího výkonu a 3. kam umístit populační paměť s konfiguračními řetězci jedinců populace.

Jelikož velikost konfiguračního řetězce se pohybuje maximálně kolem několika tisíců bitů, není problém udržovat celou populační paměť v paměti cache. Získáme tím velmi vysoký výkon a odpadne nutnost implementovat vysokorychlostní rozhraní. Během rekonfigurace *VRC* by ale bylo nutné přenášet pro každého jedince populace celý konfigurační řetězec, což vzhledem k tomu, že se používá pouze operátor mutace je velmi nevýhodné. Efektivnější by bylo, kdyby populační paměť byla umístěna v hardware a zapisovaly se pouze změněné byty. Dalším argumentem pro zavedení externí paměti je skutečnost, že při žádosti o novou konfiguraci *VRC* by se celého procesu rekonfigurace musel zúčastnit i procesor, který by posílal postupně, v nejlepším případě po 64 bitech, celou konfiguraci. Tím by došlo k prodloužení doby rekonfigurace, neboť konfigurace jednoho sloupce by trvala několik taktů. Bohužel i použití externí paměti má své nedostatky. Vytvořit novou populaci z jediného jedince by znamenalo načítat data z externí paměti a zapisovat je mírně modifikovaná zpět, pouze na jiné místo, což v případě, kdy externí paměť nepracuje na frekvenci jádra, znamená mnoho nevyužitých taktů. Nejefektivnější je oba přístupy zkombinovat, tzn. udržovat populaci duplicitně v cache i v externí paměti. Mutace by se prováděla současně v externí paměti i v cache a při tvorbě nové populace z nejlepšího jedince by se data četla z paměti cache. Teoreticky lze vystačit pouze s externí

pamětí, neboť v případě použití BRAM připojených na rozhraní *OCM* je možné dosáhnout výkonnosti ekvivalentní zásahu do cache. V praxi ovšem narazíme na problém získat design pracující na frekvenci jádra.

Na místě externí paměti máme možnost použít paměti BRAM, které připojíme na rozhraní *OCM*. Jelikož datová šířka *OCM* je 32 bitů, ale PowerPC umožňuje pracovat i s jednotlivými byty, je nutné zvážit, jak se postavit k signálu BYTEENABLE. Ten určuje, které byty slova se mají zapsat do paměti. Pokud bychom používali pouze zarovnané přístupy a pro přístup k paměti pouze instrukce pracující s celým slovem, můžeme tyto signály ignorovat a zapisovat celé slovo. Jediné efektivní řešení tohoto problému je rozložit 32 bitů do čtyř BRAM pamětí s datovou šířkou osm bitů. Tím bude mít každá BRAM vlastní signál povolení zápisu. Paměti BRAM lze nakonfigurovat na několik datových šířek (1, 2, 4, $8+1$, $16+2$ nebo $32+4$ bitů), kromě toho je možné je nakonfigurovat v DUALPORT režimu, tzn. s dvěma plně nezávislými porty (včetně hodinových vstupů). Jeden port lze připojit k *OCM* a druhý port k *PMI*, jedinou podmínkou je, aby se současně nečetlo a nezapisovalo na stejnou adresu, což lze velmi snadno zaručit.

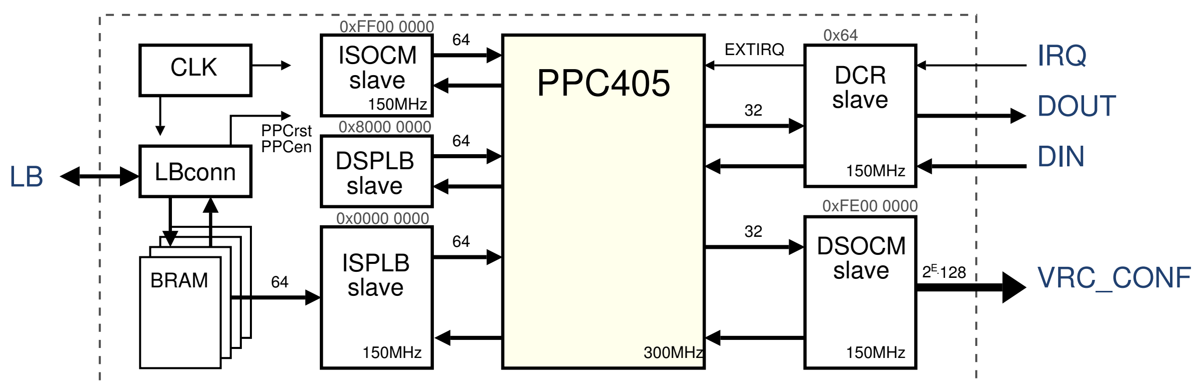
Nejjednodušším řešením by bylo na straně *PMI* použít port se stejnou datovou šířkou (osm bitů). Pokud bychom však použili jen čtyři BRAM, byl by výkon srovnatelný s řešením bez paměti, neboť by paměť konfiguraci serializovala po 32 bitech. Snahou tedy je na straně *PMI* získat co nejširší port. V nejlepším případě s šířkou odpovídající počtu bitů konfiguračního řetězce připadajícího na konfiguraci jednoho sloupce. Toho docílíme pokud použijeme větší počet BRAM pamětí, které jsou ze strany procesoru adresovány postupně po čtveřicích, z druhé strany tvoří jeden široký port. Jedná se ovšem o velmi neefektivní řešení, protože na 300 bitovou konfiguraci je zapotřebí 40 pamětí BRAM, jejichž kapacita bude téměř nevyužita. Jeden jedinec zabere v každé paměti jen tolik bitů, kolik je sloupců konfigurace. Navíc se takový design nemusí vzhledem k množství propojení podařit prakticky implementovat. Abychom jednotlivé paměti co nejvíce využili a snížili jejich počet, je nutné na druhém portu použít větší datovou šířku – 32bitů. Je ale nutné vyřešit, jak se vypořádat s nehomogenním způsobem adresace. Ze strany procesoru adresujeme po 4×8 bitech, z druhé strany po 4×32 bitech. Z hlediska evoluce se tímto problémem není nutné zabývat, neboť ta na úrovni bitů nerozlišuje, o který bit se jedná. Navíc díky možnosti ovlivnit architekturu rekonfigurovatelného obvodu *VRC* to vůbec není nutné. Vhodným návrhem lze totiž zajistit, aby jakýkoliv konfigurační řetězec byl platný a nevedl k chybným výsledkům. Jelikož je ale cílem vytvořit obecnou architekturu, musíme se s tímto problémem vypořádat. Řešením je neadresovat na straně procesoru pamět sekvenčně, ale jiným způsobem. To je však nejhorší možná varianta, neboť se jedná o pouhé prohození bytů, což je na úrovni hardware, narozdíl od software, operace, která nic nestojí.

V následujících odstavcích se budeme podrobněji věnovat postupně každému z bloků architektury uvedené na obrázku 37. Ke každému složitějšímu bloku bude uvedeno i zjednodušené schema, které obsahuje jen nejdůležitější komponenty.

9.3 Blok PPC

Blok PPC je nejsložitějším blokem celého systému. Jeho úkolem je zajistit chod procesorového jádra a poskytovat rozhraní jednotce *PMI*. Blok tvoří procesorový blok *PPC405* a řada periferií

nezbytných pro činnost procesoru – paměti, řadiče sběrnic a řídicí logika. Architektura celého bloku je uvedena na obrázku 39.



Obrázek 39: Blokový diagram bloku PPC. U každé periferie je pomocí bázové adresy naznačeno mapování do adresového prostoru a pracovní kmitočet. Datová šířka uvedená u sběrnic je pouze pro orientaci.

Protože cílem je navrhnout systém maximálně využívající dostupné zdroje v FPGA, je snahou, aby procesor pracoval na maximální možné frekvenci. Uvnitř FPGA obvodu, kterým je osazena karta Combo6X, je možné procesor provozovat až na frekvenci 300 MHz. V případě použití FPGA stejného typu, avšak s nižším „speed grade“ číslem[2], je možné dosáhnout frekvence až 400 MHz. Varianta, kdy bylo pro instrukční i datovou paměť použito rozhraní *OCM*, se v praxi neosvědčila, neboť není reálné naroutovat design na 300 MHz. S taktovacím kmitočtem rozhraní *OCM* se muselo přejít na poloviční kmitočet. To však vede na poloviční výkonnost, neboť jak jsme uvedli dříve, není možné na prostor mapovaný pomocí *OCM* zapnout cache. Z toho důvodu je zapotřebí použít komunikačně složitější, ale ve výsledku výkonnější sběrnici *PLB*. Taktovací kmitočet sice musí být opět ze stejného důvodu poloviční oproti frekvenci jádra, narozdíl od *OCM* je však možné zapnout podporu cache. Kapacita paměti cache je dostačující jak pro data tak pro instrukce.

Bloky *ISOCM* a *ISPLB* realizují rozhraní mezi instrukční pamětí a řadičem instrukční cache, *DSPLB* rozhraní mezi datovou pamětí a řadičem datové cache. Řadič připojený na sběrnici *DCR* umožnuje komunikaci s okolím. Pomocí *DSOCM* je do datové paměti namapována populační paměť, která je navíc zpřístupněna ve formě širokého portu jednotce *PMI*.

Aby bylo možné procesor programovat ze strany PC, je instrukční paměť připojena na sběrnici LocalBus (LB). Paměť je ze strany PC přístupná pro čtení i zápis. Na sběrnici LB je připojen i řídicí registr, pomocí kterého je možné řídit signály sloužící k inicializaci (PPCRST) a povolení činnosti procesoru (PPCEN). Jednotlivým blokům uvedeným na obrázku 39 se detailně věnují následující podkapitoly.

[2]index udávající výkonnostní třídu FPGA obvodu. Čím nižší je toto číslo, tím vyšší je maximální pracovní kmitočet bloků uvnitř FPGA

9.3.1 CLK

Blok CLK se stará o generování hodinových signálů pro procesorové jádro a jednotlivá rozhraní. Protože je k dispozici pouze jeden externí hodinový signál o frekvenci 50 MHz, je nutné veškeré hodinové signály získat odvozením od tohoto kmitočtu. K tvorbě hodinových signálů slouží konfigurovatelné jednotky DCM pracující na principu fázového závěsu. DCM jsou distribuovány po obvodu čipu FPGA, aby spojení s externím hodinovým signálem bylo co nejkratší. Tyto jednotky jsou schopny generovat široké spektrum frekvencí, umožňují frekvenci nejen snížit, ale i zvýšit.

DCM jednotka odvozuje výstupní kmitočet od kmitočtu vstupního signálu CLKIN v závislosti na zvoleném poměru. Všechny výstupní signály získané pomocí DCM jsou fázově zarovnané. V aplikaci je možné použít libovolnou kombinaci následujících výstupů:

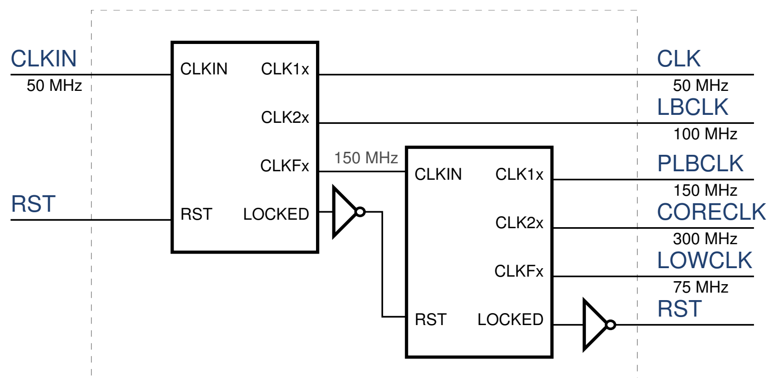
- CLK1x – výstup stejné frekvence jako vstupní signál, $f_{CLK1x} = f_{CLKIN}$,
- CLK2x – výstup s dvojnásobnou frekvencí, $f_{CLK2x} = 2f_{CLKIN}$,
- CLKFx – výstup s frekvencí určenou poměrem $M : D$, $f_{CLKFx} = \frac{M}{D} \cdot f_{CLKIN}$,
- CLKDV – výstup o frekvenci zmenšené v poměru $1 : V$, $f_{CLKDV} = \frac{1}{V} \cdot f_{CLKIN}$.

Aby fázový závěs, který se používá pro generování signálu CLKFx, pracoval spolehlivě, je nutné hodnoty násobitele M a dělitele D volit tak, aby nebyl překročen frekvenční rozsah garantující stabilitu. Frekvenční rozsahy pro konkrétní FPGA (speedgrade 5) a oba DCM režimy jsou uvedeny v následující tabulce.

Režim DCM	CLKIN	CLKFx
Low Frequency Mode	1-210 MHz	24-210 MHz
High Frequency Mode	50-270 MHz	210-270 MHz

Tabulka 2: Frekvenční rozsah CLKFx a CLKIN v závislosti na zvoleném režimu

Z hodnot uvedených v tabulce je zřejmé, že pro 50 MHz vstupní signál nejsme schopni v žádném režimu vygenerovat na výstupu CLKFx signál o kmitočtu 300 MHz. Musíme proto použít dva DCM bloky v sérii a pro 300 MHz signál použít výstup CLK2x, jak ukazuje následující obrázek.



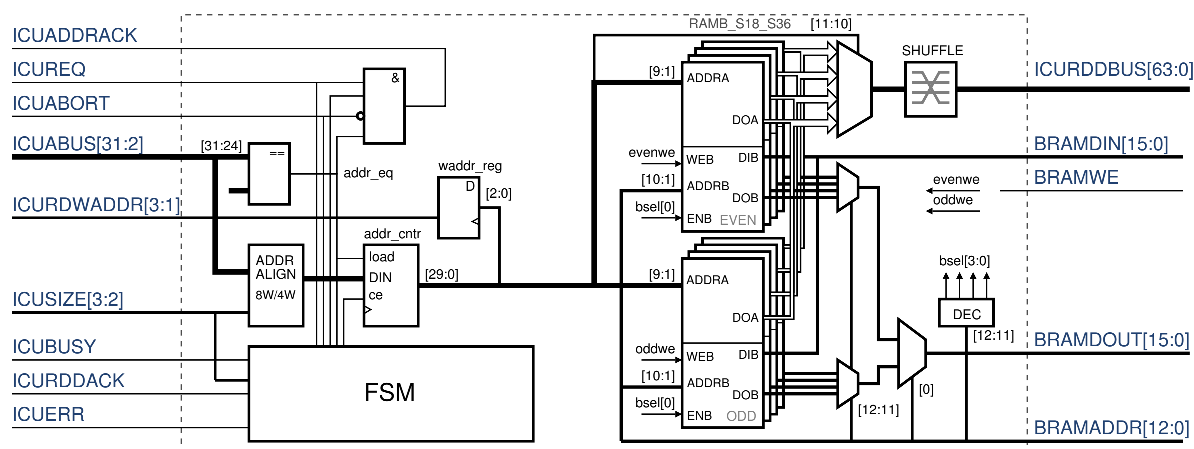
Obrázek 40: Blokový diagram CLK jednotky

9.3.2 IS PLB

Primárním úkolem bloku *ISPLB* je zajistit rozhraní mezi pamětí programu tvořenou sadou pamětí BRAM a řadičem instrukční cache. Dalším úkolem je dát k dispozici rozhraní, které umožní přistupovat k paměti pomocí sběrnice LocalBus.

Kapacita instrukční paměti byla zvolena 16kB, což odpovídá velikosti cache. Celý řadič je však napsán genericky a umožňuje kapacitu libovolně rozšířit. Protože instrukční *PLB* vyžaduje 64 bitovou datovou šířku, je paměť programu organizována po dvojicích BRAM s 32 bitovou datovou šířkou – tzv. bank. Kapacitu je možné zvýšit zvýšením počtu banků. Počet banků musí být roven mocnině dvou, aby je bylo možné efektivně adresovat.

Blok je navržen jako 64 bitový *ISPLB* slave, který z důvodu zjednodušení nepodporuje řetězení adresy. Uvedené zjednodušení však není na závadu. Řetězení by bylo nutné řešit pouze v případě, kdy by kapacita instrukční paměti přesahovala kapacitu cache nebo bychom cache vůbec nevyužívali. Schéma je uvedeno na obrázku 41.



Obrázek 41: HW realizace instrukční paměti připojené k *ISPLB*

Činnost celého zařízení řídí konečný automat FSM, který je aktivován pouze tehdy, vznikne-li požadavek na čtení a prvních osm bitů adresy je shodných s genericky zvolenou konstantou. K potvrzení žádosti o čtení ICUREQ slouží kombinačně generovaný signál ICUADDRACK, který je generován pouze tehdy, neprobíhá-li právě čtecí transakce (zákaz řetězení adresy). Kombinačně generované potvrzení umožňuje zahájit transakci hned v následujícím taktu. Adresa požadované instrukce ICUABUS vstupuje do bloku zarovnání adresy, který v závislosti na režimu (čtení slova, čtení řádky cache) provádí zarovnání na čtyři nebo osm slov. Počet čtených slov určuje sběrnice ICUSIZE. Po inicializaci procesoru je implicitním režimem čtyřslovní transakce, je-li zapnuta podpora cache, čte řadič cache z *PLB* po osmi slovech. Jelikož jedno slovo odpovídá 32 bitům a datová šířka *ISPLB* odpovídá dvěma slovům, trvá čtení čtyřslova dva takty PLBCLK a čtení řádky cache čtyři takty.

V taktu bezprostředně následujícím po potvrzení žádosti je do adresového čítače addr_cntr uložena adresa ICUABUS, která slouží k adresaci paměti BRAM. Po přenosu jednoho 64 bitového slova je adresa čítače zvýšena, aby ukazovala na adresu následující dvojice instrukcí.

Protože je BRAM synchronní, je nutné počítat s latencí jeden takt, která se projeví vznikem prázdného taktu mezi potvrzením žádosti a přenosem prvního slova. Během celé transakce je aktivní signál ICUBUSY. Signál ICURDDACK je aktivní pouze tehdy, obsahuje-li datová sběrnice ICURDDBUS platné slovo.

K připojení na LocalBus slouží rozhraní, které poskytuje plný 16 bitový přístup k instrukční paměti. Datová šířka 16 bitů byla zvolena záměrně, neboť rozhraní připojené na sběrnici LocalBus pracuje oproti sběrnici na dvojnásobném kmitočtu (100 MHz) a poloviční datové šířce (16 bitů). Nejefektivnějším řešením je použít paměť BRAM s různou šířkou portů (32 bitů pro *PLB* a 16 bitů pro LB). Tím však dochází k přehazování slov, se kterým je potřeba se vypořádat buď na straně PC, na straně portu BRAM, nebo na straně datové sběrnice ICURDDBUS. Abychom mohli ze strany PC (přes LocalBus) zapisovat slova v přirozeném pořadí, je přehození slov řešeno blokem SHUFFLE, který je připojen na 64 bitový datový výstup ICURDDBUS. Účelem bloku je zajistit, aby zápisem čtyř šestnáctibitových slov vznikly dvě 32 bitové instrukce (např. zápisem slov 1122 3344 5566 a 7788 vzniknou dvě instrukce tvaru 11223344 a 55667788). Ukazuje se, že stačí pouze zaměnit nejméně významné a nejvýznamější slovo, jak je schematicky znázorněno ve schématu.

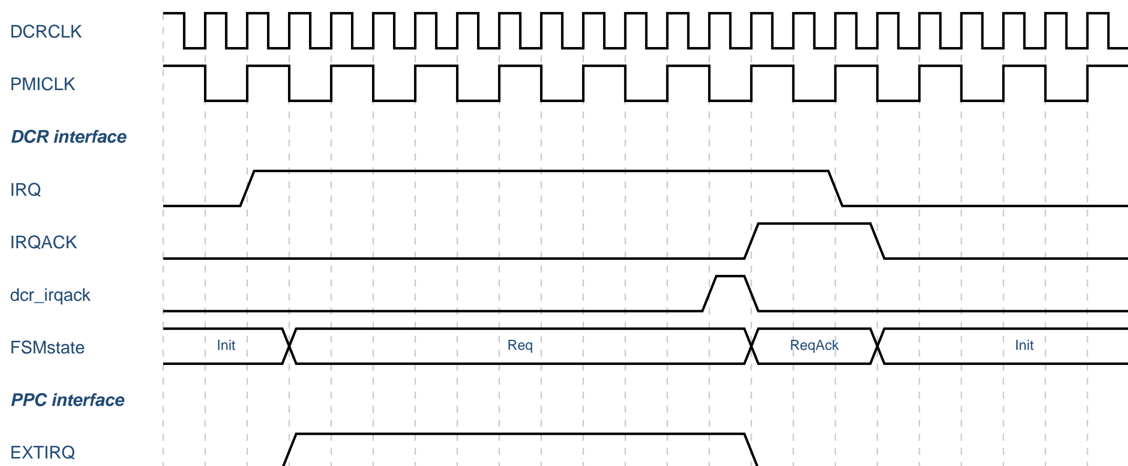
9.3.3 DS PLB

Blok *DSPLB* tvoří rozhraní mezi 16kB datovou pamětí a řadičem datové cache. Jeho struktura je velmi podobná řadiči *ISPLB* s tím rozdílem, že je zapotřebí řešit i zápisy, které mohou být nezarovnané. Fyzicky ale není řadič připojen k žádné paměti. Využíváme skutečnosti, že na datovou paměť stačí 16kB a že je možné na celou datovou oblast povolit práci s cache. Software při inicializaci přepne aktualizaci paměti do režimu write-back, čímž se odstraní přenosy po sběrnici vznikající zápisem do paměti cache. Řadič *DSPLB* vrací řadiči cache stále nulovou hodnotu, čehož je možné využít v aplikaci, kde není nutné neinicializovanou paměť inicializovat na nulovou hodnotu. Při psaní programu je nutné mít stále na paměti skutečnost, že není možné používat staticky inicializované proměnné. Staticky inicializované proměnné by byly mírnou komplikací i v případě, že bychom datovou paměť realizovali, neboť by bylo nutné přeložený strojový kód rozdělit na nezávislou datovou a instrukční oblast.

9.3.4 DCR

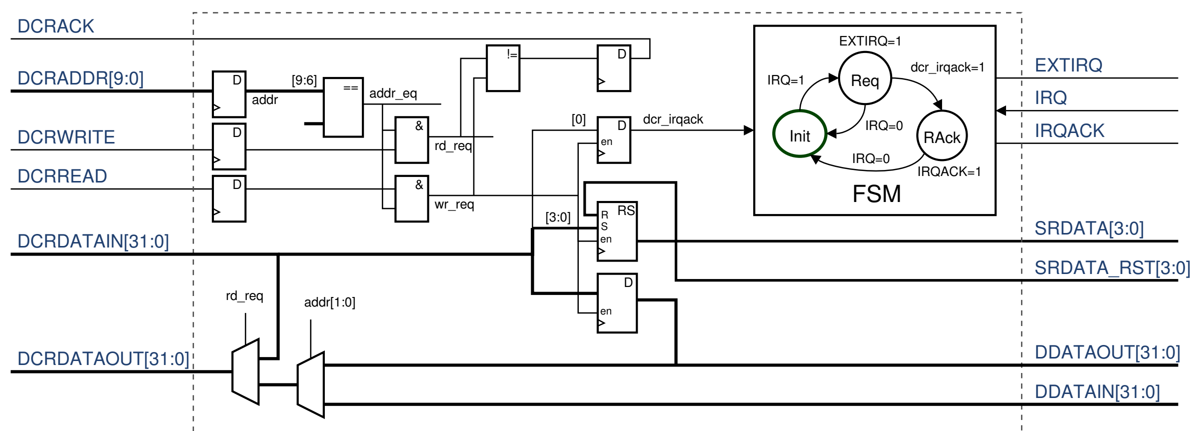
Rozhraní *DCR* slouží ke komunikaci s jednotkou *PMI* a k řízení externího přerušení. Základ řadiče přerušení tvoří jednoduchý automat se třemi stavy. Při detekci žádosti o přerušení ze strany hardware (aktivní signál IRQ) je nastaven signál EXTIRQ, který je přiveden na vstup procesoru. Tento signál setrvá aktivní do té doby, než je přerušení vyřízeno a potvrzeno zápisem do registru *DCR*. Tehdy je aktivován signál IRQACK, který indikuje vyřízení žádosti a má za následek deaktivaci žádosti o přerušení ze strany hardware. Průběh komunikace je znázorněn na diagramu 42.

Kromě řízení přerušení umožňuje *DCR* z číst a zapisovat 32 bitové slovo. Aby nebylo nutné řešit přístup přes třístavový buffer, je slovo rozděleno do dvou sběrnic – vstupní DDATA_IN a výstupní DDATA_OUT. Čtení slova slouží k získání fitness hodnoty vystavené na sběrnici DDATA_IN. Zápisu do registru připojeného na sběrnici DDATA_OUT se využívá k řízení.



Obrázek 42: Průběh vyřízení žádosti o přerušení pomocí řadiče přerušení

Kromě 32 bitového vstupu a výstupu obsahuje *DCR* speciální registr SRDATA, který je možné ze strany *PMI* jednotky po bitech nulovat a ze strany software po bitech nastavovat. Tento registr informuje *PMI* o stavu konfiguračních řetězců uložených v populační paměti. Počet bitů registru odpovídá maximálnímu počtu řetězců. Logická jednička znamená, že konfigurační řetězec je platný a je možné ho použít pro evaluaci.



Obrázek 43: HW realizace bloku *DCR*

9.3.5 IS OCM

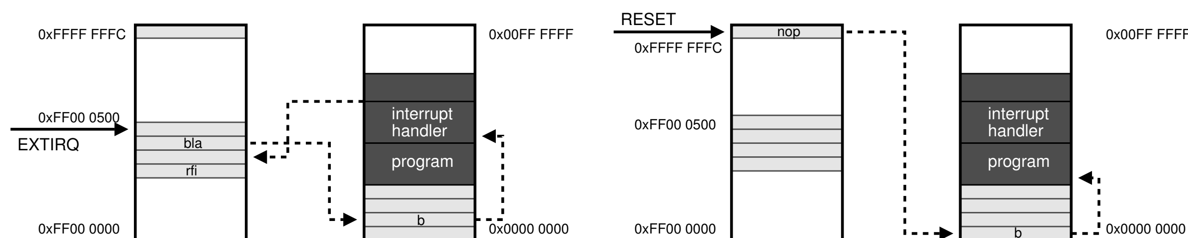
Důvodem k zavedení instrukční paměti přístupné přes rozhraní *OCM* je efektivnější využití adresového prostoru. V kapitole 8.1.2 věnující se inicializaci procesoru jsme se zmínili, že první instrukce prováděného kódu musí být umístěna na adrese 0xFFFF FFFC. Jelikož vystačíme s 16kB instrukční pamětí, bylo by nutné umístit celou instrukční paměť do oblasti 0xFF00 0000,

konkrétně od adresy 0xFFFB FFFC. Komplikace přináší potřeba podporovat obsluhu přerušení. Způsob, jakým PowerPC určuje adresu obsluhy přerušení, je bohužel nevhodný pro aplikace, které obsahují malé množství instrukční paměti. Adresa obsluhy přerušení je určena pomocí bázové adresy a posunutí. Problematické je nepravidelně rozmístěné posunutí, které je navíc velmi vysoké – např. 0x500 pro externí přerušení. Protože linker neumí sestavit kód tak, že by přerušil souvislý blok instrukcí a vložil na určitou adresu obslužnou rutinu, je nutné dát obsluhu přerušení na pevnou adresu nebo až za konec souvislého bloku. Obě varianty však znamenají množství nevyužitého místa.

Možností, jak problém vyřešit, je připojit na *ISPLB* další zařízení, které by hardwarově řešilo rozptýlenou obsluhu přerušení. To ovšem značně komplikuje celý systém, neboť by bylo nutné implementovat i *PLB* arbitr. Mnohem jednodušší je použít ke stejnému účelu doposud nevyužité rozhraní *ISOCM*.

ISOCM je namapováno do oblasti 0xFF00 0000–0xFFFF FFFF. Úkolem je poskytnout prostředek, kterým by bylo možné implementovat klasickou tabulku přerušení. Uvedená oblast byla zvolena záměrně, aby bylo možné obsloužit i první instrukci. Řešení je čistě kombinační a skládá se pouze z několika komparátorů připojených na adresovou sběrnici a multiplexoru, který přepíná na výstup požadovaný instrukční kód.

Abychom linearizovali rozptýlené adresy obslužných rutin, byla implementována dvouúrovňová obsluha přerušení. V paměti programu mapované od adresy 0x0000 0000 je rezervováno několik prvních slov pro vektory přerušení. Vektor přerušení je složen z instrukce větvení **b** (*branch*) na obslužnou rutinu. Bázová adresa přerušení je nastavena pomocí registru EVPR na 0xFF00 0000. Pokud přijde např. externí přerušení, začne procesor provádět instrukce od adresy 0xFF00 0500. Zde je umístěn kód, který přemapuje skok např. na adresu 0x0000 0004. Obsluha přerušení je ilustrována obrázkem 44. Výhodou je, že adresy v tabulce vektorů přerušení je schopen překladač dopočítat (vloží se pouze název funkce), obsluhu přerušení (interrupt handler) proto může tvořit libovolná funkce umístěná na libovolné adrese.



Obrázek 44: Princip obsluhy externího přerušení (vlevo) a inicializace (vpravo) pomocí tabulky vektorů přerušení

Jelikož je přerušení asynchronní událost, musí rutina provádějící přemapování, manipulovat s registry tak, aby nedošlo k jejich přepsání. To by mohlo po návratu z přerušení způsobit chybu. K návratu z obslužné rutiny se využívá standartní návrat pomocí příkazu return, který je implementován pomocí skoku na adresu určenou link registrem. Uložení návratové adresy před každým voláním funkce zajišťuje překladač.

Rutina obsažená v HW provádí následující kód:

```
9421FFF0 stwu r1, -16(r1)     //rezervace prostoru na zásobníku
9001000C stw  r0,  12(r1)     //uložení registru r0
7C0802A6 mflr r0
90010008 stw  r0,  08(r1)     //uložení link registru
48000007 bla  04              //skok do tabulky přerušení na 2. položku

80010008 lwz  r0,  08(r1)
7C0803A6 mtlr r0              //obnovení link registru
8001000C lwz  r0,  12(r1)     //obnovení registru r0
38210010 addi r1,  r1, 16     //uvolnění prostoru v zásobníku
4C000064 rfi                  //návrat z přerušení
```

Před skokem do tabulky vektorů přerušení je zapotřebí uložit link registr, neboť bude modifikován. Na zásobníku se vytvoří prostor, do kterého jsou uloženy modifikované registry. Poté se pomocí instrukce **bla** (*Branch Absolute and Link*) provede skok do tabulky vektorů přerušení. Výhodou této instrukce je, že před skokem uloží do link registru adresu následující instrukce. Obslužných rutin je zapotřebí tolik, kolik vyžadujeme podporovat přerušení. Jednotlivé rutiny se však liší pouze v adrese použité na pátém řádku.

Struktura tabulky vektorů přerušení je následující:

```
48000010 b reset               //skok na první instrukci programu
4800004c b external_interrupt  //skok na obsluhu externího přerušení
48000050 b critical_interrupt  //skok na obsluhu kritického přerušení
48000000 b $                   //nevyužité přerušení
```

Tabulka vektorů přerušení musí obsahovat pouze instrukce, které neovlivňují stav procesoru, především stav link registru, který obsahuje návratovou adresu. Vhodná instrukce je instrukce větvení **b** (*branch*), která nemění žádný registr. Význam jednotlivých položek určují adresy v HW obslužných rutinách.

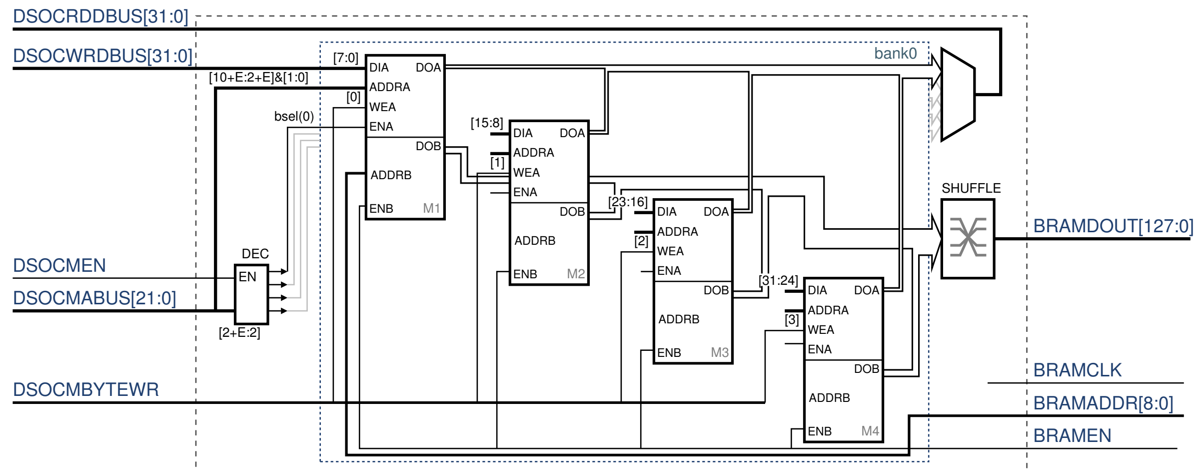
9.3.6 DS OCM

Jak již bylo uvedeno dříve, slouží rozhraní *DSOCM* k zpřístupnění externí populační paměti, kterou tvoří sada pamětí BRAM. Přestože jsme se využití rozhraní *OCM* doposud, především z důvodu nižší výkonnosti, vyhýbali, jeví se *OCM* v tomto kontextu jako nejschůdnější varianta. Připomeňme, že nižší výkonnost je způsobena nutností pracovat na polovičním taktovacím kmitočtu a nemožností povolit nad mapovanou oblastí využívat cache.

Nyní je ale situace odlišná. Tím, že z externí paměti může číst i jednotka *PMI*, je žádoucí, aby veškerá data byla zapisována přímo do BRAM a neprocházela skrz paměť cache. Kromě *OCM* je samozřejmě možné využít ke stejnému účelu i datovou sběrnici *PLB*, neboť ta dovoluje nad mapovanou oblastí cache vypnout. Navíc díky řetězení adresy lze získat přibližně dvakrát vyšší výkonnost. Ovšem připojit na datovou *PLB* další zařízení by byla zbytečná komplikace vyžadující implementaci *PLB* arbitru.

Abychom dosáhli maximálního využití kapacity a současně širokého datového portu na straně jednotky *PMI*, jsou použity dvouportové paměti s rozdílnými šířkami jednotlivých portů (8 a 32 bitů). Paměti jsou organizovány po bancích, které tvoří čtveřice BRAM. Na jeden bank jsme schopni na straně *PMI* dosáhnout datové šířky 128 bitů ($32 \cdot 4$). V případě, že počet bitů konfiguračního řetězce převyšuje 128 bitů, je nutné přidat další bank. Počet paměťových

banků musí být roven mocnině dvou, aby bylo možné banky efektivně adresovat. Schéma uvedené na obrázku 45 znázorňuje z důvodu přehlednosti pouze jeden bank. Celý řadič je však implementován genericky a umožňuje parametricky stanovit počet banků a tím určit výslednou šířku portu. Šířka je rovna $128 \cdot 2^{\lceil log_2 b \rceil}$, kde b je počet banků. Protože počet bitů portu je v násobcích 128, může získaný počet bitů převyšovat požadavky *VRC*. Takové bity jsou sice ze strany procesoru k dispozici, nejsou však nikam zapojeny.

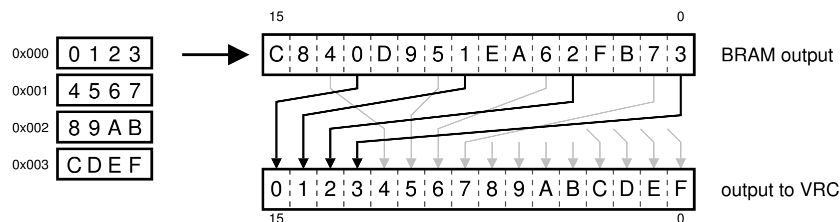


Obrázek 45: HW realizace bloku *DSOCM*. Kvůli přehlednosti je zkreslen pouze jeden bank a kompletní cesty pouze k první paměti. Signály závislé na počtu banků jsou parametrizovány konstantou E. Platí $b = 2^E$, kde b je počet bank.

Uvedená organizace umožňuje pojmout konfigurační řetězec, který je složen z libovolného počtu řádků a maximálně 512 sloupců. Na jeden bank tvořený čtyřmi BRAM a celkovou kapacitou 8kB připadá právě pětsetdvanáct 128 bitových řetězců. Přidáním dalšího banku sice vzroste celková kapacita na dvojnásobek, zároveň však dvojnásobně vzroste i počet bitů širokého portu. Dovolíme-li maximálně 16 sloupců *VRC* připadajících na jednoho jedince, je schopna paměť pojmout 32 jedinců populace.

Podobně jako tomu bylo u bloku *ISPLB*, dochází i zde vlivem rozdílné datové šířky jednotlivých portů k prohazování na úrovni bytů. Aby bylo možné z procesoru zapisovat data ve standartním pořadí, řeší se prohazování až na straně širokého portu. Schematické znázornění bloku realizujícího přeuspořádání šestnácti bytů je na obrázku 46.

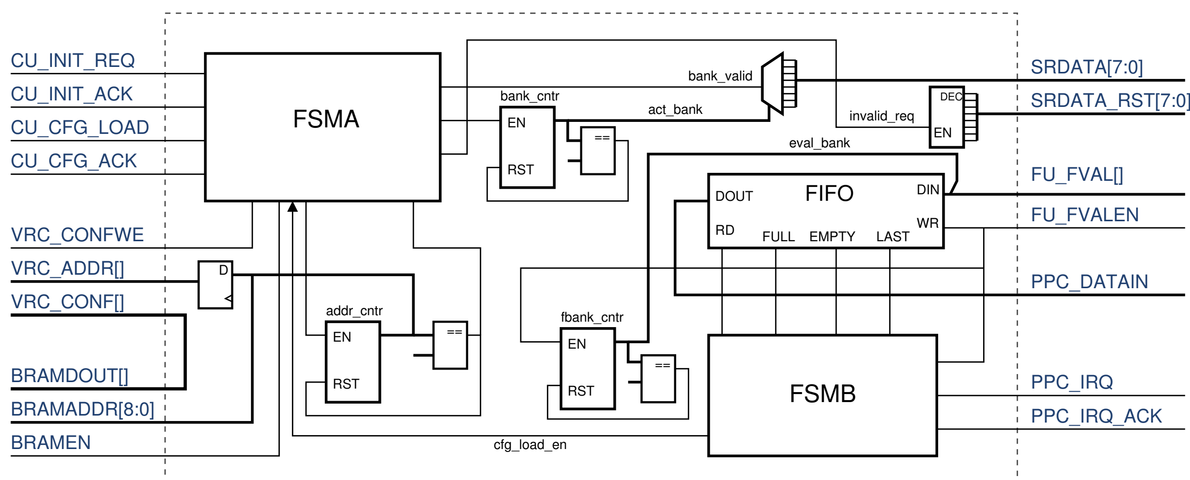
Jelikož blok *OCM* využívá rozhraní *DSOCM* taktované polovičním kmitočtem, dosahuje se maximálně poloviční přenosové kapacity. V tomto případě to ale není na závadu, neboť procesor má na zápis konfiguračního řetězce poměrně dlouhou dobu. Zápis totiž probíhá současně s evaluací. To dovoluje plné překrytí obou fází bez nutnosti čekat. Ke snížení propustnosti dochází pouze tehdy, je-li doba evaluace kratší, než doba potřebná k zápisu konfigurace do paměti BRAM. V praxi jsou běžnější takové případy, kdy doba evaluace je vzhledem k době zápisu mnohem větší.



Obrázek 46: Bytové přeuspořádání 128 bitů. Cílem je, aby zápis čtyř po sobě jdoucích 32bitových slov ze strany procesoru (vlevo) tvořil na výstupu jedno 128bitové slovo, ve kterém je zachováno pořadí.

9.4 Blok PMI

Tento blok tvoří most mezi hardwarovou a softwarovou částí evolučního algoritmu. Skládá se ze dvou nezávislých paralelně pracujících celků. První část, řízená jednotkou *CU*, se stará o rekonfiguraci *VRC* obvodu – po sloupcích zapisuje konfigurační řetězec uložený v populační paměti do *VRC*. Druhá část, řízená fitness jednotkou *FU*, zajišťuje přenos vypočtené fitness hodnoty do procesoru. Mimoto blok poskytuje rozhraní k populační paměti, které je možné připojit na sběrnici LocalBus. Jelikož jsou zapotřebí tři nezávislé přístupové body a BRAM má pouze dva porty, je nutné jeden z portů multiplexovat. Multiplex se provádí na portu vedoucím do *VRC*, neboť na druhém portu by byl z důvodu dosti vysokého pracovního kmitočtu problematický.



Obrázek 47: Schéma bloku *PMI*. Z důvodu přehlednosti není zakreslena část realizující rozhraní pro LocalBus.

Podrobnější schema celého bloku je uvedeno na obrázku 47. Srdce tvoří dva nezávislé stavové automaty. Jelikož potřebujeme snadno synchronizovat činnost procesoru a hardware, je populační paměť rozdělena na několik oblastí – tzv. banky (nesouvisí s banky uvedenými dříve). Každý bank obsahuje jeden konfigurační řetězec a je ohodnocen nezávisle na ostat-

ních. Na adresaci banků je využito horních B bitů ($B = \lceil \log_2 bankcount \rceil$) adresní sběrnice BRAMADDR. Každému banku přísluší bit, který určuje, jsou-li data platná a lze-li je použít na evaluaci. Počet banků nemusí vždy odpovídat počtu jedinců populace, záleží na konkrétní implementaci evolučního algoritmu. Smyslem rozdělení je umožnit překrýt fázi ohodnocení s fází generování nové konfigurace. Úplného překrytí dosáhneme již při použití dvou banků. Po ohodnocení konfigurace uložené v prvním banku se plynule naváže na ohodnocení konfigurace uložené v druhém banku. Současně s evaluací druhé konfigurace lze do prvního banku nahrávat nový řetězec a tím po skončení evaluace opět plynule navázat. Čím více použijeme banků, tím větší prostor dáváme procesoru k vytvoření nové generace. Po vyčerpání maximální doby však musí být procesor schopen relativně rychle generovat jednotlivé jedince, jinak vzniknou nevyužité takty. Počet banků je omezen tím, že počet sloupců, který máme možnost naadresovat v rámci jednoho banku, musí odpovídat minimálně počtu sloupců konfiguračního řetězce. V našem případě je populační paměť rozdělena na osm banků. Na jeden řetězec tedy připadá maximálně 64 sloupců.

Činnost celého bloku lze shrnout následovně. Řídicí jednotka požádá o rekonfiguraci *VRC* aktivací signálu CU_CFG_LOAD. V případě, že aktuální banka obsahuje platná data (bank_valid), je žádost potvrzena signálem CU_CFG_ACK a začne se s konfigurací *VRC*. Po jednom taktu, kdy došlo k nakonfigurování prvního sloupce, aktivuje řídicí jednotka fitness jednotku, která zahájí ohodnocení. Během konfigurace posledního sloupce se zneplatní bank nastavením signálu inval_req a zvýší se hodnota čítače bank_cnt, který určuje aktuálně zpracovávaný bank. Pokud není obsah aktuálního banku validní, je evaluace pozastavena do té doby, než dojde k jeho aktivaci.

Fitness jednotka indikuje konec evaluace aktivací signálu FU_FVALEN, který uvede v činnost automat FSMB. Jen dodejme, že ke konfiguraci *VRC* novým konfiguračním řetězcem však dochází o takt dříve, aby bylo možné ihned po skončení předešlé evaluace navázat další bez vzniku prázdných taktů. Vlivem platného signálu FU_FVALEN dojde k uložení stavu, který se skládá z vypočtené fitness hodnoty a pořadového čísla vyhodnoceného banku. Současně generuje automat žádost o přerušení (pokud se právě některá z předešlých žádostí nevyřizuje), která aktivuje příslušnou obslužnou rutinu. Ta zajistí načtení uloženého stavu pomocí *DCR*, vygenerování nové konfigurace do banku ve kterém došlo k evaluaci, nastavení valid bitu a potvrzení přerušení. Abychom pomocí relativně pomalého rozhraní *DCR* nemuseli zapisovat několik slov, jsou bity udávající platnost banku a potvrzení přerušení sloučeny do jednoho 32bitového slova.

Automat řídící přerušení umožňuje uložit tolik stavů (výsledků), kolik je celkový počet banků. Pokud je zaplněna předposlední položka, dojde k zablokování další evaluace, aby nemohl vzniknout další stav, který by nebylo možné uložit. K uschování stavů se používá paměť FIFO, jejíž výstupní část je řízena automatem a vstupní část signálem FU_FVALEN. Fronta umožňuje provést evaluace všech banků, aniž by procesor musel dokončit obsluhu prvního přerušení.

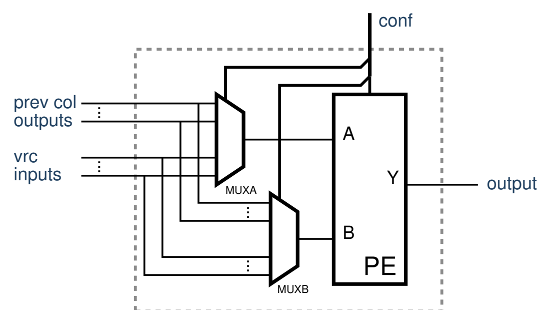
9.5 Blok VRC

Virtuální rekonfigurovatelný obvod, který budeme používat, odpovídá popisu uvedenému v kapitole 5.3.3. Zatím jsme se nezmínili o tom, jaká je vlastně šířka konfiguračního řetězce

připadající na jeden sloupec. Vyjdeme ze základní struktury elementu, uvedené na obrázku 48. Abychom plně popsali konfiguraci jednoho elementu, je zapotřebí určit, kam je připojen první a druhý vstup programovatelného elementu PE a jakou bude realizovat funkci. Architektura *VRC* dovoluje připojit vstupy buď na výstupy elementů předchozího sloupce nebo na vstupy *VRC*. Počet elementů předchozího sloupce odpovídá počtu řádků *VRC*. Na konfiguraci jednoho elementu je zapotřebí w_e bitů, kde w_e je rovno

$$w_e = 2\lceil \log_2(vrc_inputs + vrc_rows)\rceil + \lceil \log_2(vrc_functions)\rceil)$$

Na konfiguraci jednoho sloupce tedy potřebujeme $w = vrc_cols \cdot w_e$ bitů.



Obrázek 48: Struktura základního elementu rekonfigurovatelného pole *VRC*

Má-li *VRC* obvod např. 16 vstupů, 16 řádků, 10 sloupců a programovatelný element může realizovat jednu z osmi funkcí, pak konfigurace jednoho sloupce zabere 208 bitů. Počet bitů konfigurace připadající na jeden sloupec se tedy v reálných aplikacích pohybuje kolem stovek bitů.

9.6 Blok FU

Posledním blokem, o kterém jsme se zatím nezmínili, je fitness jednotka. Jelikož způsob výpočtu fitness hodnoty je závislý na typu řešeného problému, uvedeme několik možností realizace, které vychází z běžně používaných metod. Fitness jednotku typicky tvoří generátor bitové posloupnosti a blok realizující výpočet fitness hodnoty. Ve většině případů generátor generuje všech 2^n možných n-bitových binárních kombinací, které jsou přiváděny na vstup rekonfigurovatelného obvodu. Vyžaduje-li aplikace test jen některých hodnot, používají se generátory založené na paměti (BRAM, SRAM, apod.). Druhou část, která zajišťuje výpočet fitness hodnoty, lze implementovat několika způsoby.

V nejjednodušším případě se používá komparátor, který porovnává vypočtenou hodnotu s požadovanou hodnotou a vrací logickou jedničku, jsou-li hodnoty rozdílné. Výstup komparátoru je připojen na vstup sčítačky, která provádí sumaci přes všechny vstupní kombinace. Na konci evaluace určuje výstup sčítačky počet chybných hodnot. Cílem evolučního algoritmu je tuto hodnotu minimalizovat na nulu. Dodejme, že sčítačka je vlastně prostý čítač, který zvýší svoji hodnotu v případě, že výstup komparátoru je aktivní.

Mírně složitějším řešením je nahradit komparátor odečítačkou, která dává absolutní rozdíl mezi vypočtenou a požadovanou hodnotou. Dá se říci, že v tomto případě je fitness hodnota

definována mnohem jemněji. Cílem evolučního algoritmu je podobně jako v předchozím případě minimalizovat získanou sumu na nulu. Tato metoda je v podstatě hardwarově realizovaná střední absolutní odchylka, která je definována následujícím vztahem.

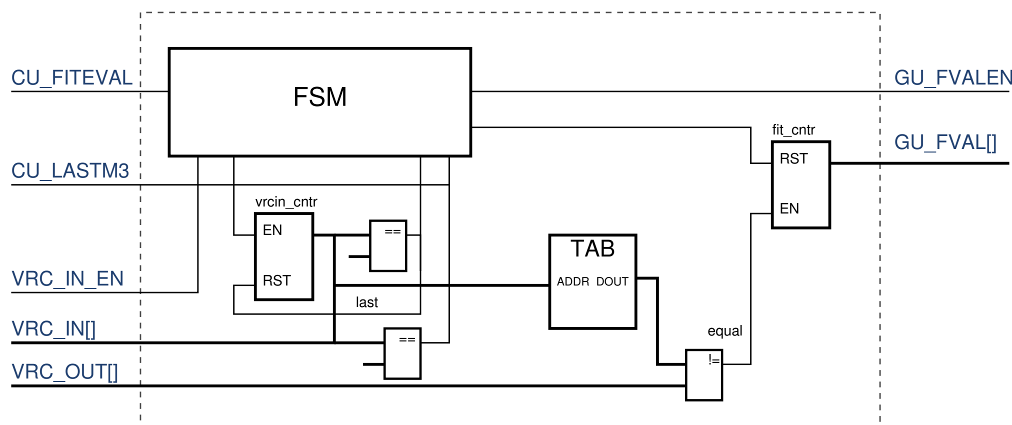
$$E_{MAE} = \frac{1}{N}\sum_{i=0}^{N}|y_i - d_i|, \tag{9.6.10}$$

kde y_i udává skutečnou a d_i požadovanou hodnotu. Jediný rozdíl spočívá v tom, že nedochází k normalizaci. V případě evoluce však normalizace nemá žádný význam, neboť počet testovacích vektorů je přes všechny jedince konstantní.

Poslední běžně používaná metoda je založena na výpočtu střední kvadratické odchylky, která je dána následujícím předpisem.

$$E_{MSE} = \frac{1}{N}\sqrt{\sum_{i=0}^{N}(y_i - d_i)^2} \tag{9.6.11}$$

Z hlediska evoluce je opět normalizace i odmocnění zbytečné. Jelikož jsou ale praktické výsledky velmi podobné jako u předchozí metody, není důvod tento způsob výpočtu používat. Oproti předchozímu případu je totiž vyžadována násobička a je nutné pro fitness hodnotu použít dvojnásobný počet bitů.



Obrázek 49: Blokové schéma fitness jednotky

Na obrázku 49 je uvedena implementace fitness jednotky, která vychází z první uvedené metody. Výpočet fitness se provádí pouze je-li signál CU_FITEVAL aktivní. Generátor generuje postupně pomocí čítače všechny n-bitové kombinace, které jsou přiváděny na vstup rekonfigurovatelného obvodu. Protože rekonfigurovatelný obvod pracuje v řetězeném režimu, je možné během jednoho taktu získat odezvu na jednu vstupní kombinaci. Aby bylo možné plynně navázat dalším výpočtem fitness, aktivuje fitness jednotka tři takty před koncem evaluace aktuálního obvodu signál CU_LASTM3. Na tento signál reaguje řídicí jednotka, která zahájí novou rekonfiguraci. Jakmile je ohodnocení ukončeno, je aktivován signál GU_FVALEN a sběrnice GU_FVAL obsahuje vypočtenou fitness hodnotu.

Odezva rekonfigurovatelného obvodu na konkrétní vstupní kombinaci je přivedena sběrnicí VRC_OUT na vstup komparátoru, který ji porovná s požadovanou hodnotou. Požadované hodnoty jsou uloženy v tabulce, která se pro malý počet položek implementuje pamětí BRAM, pro větší počet pamětí SRAM. Na rozdíl od software může být tabulka implementována i čistě hardwarově – má-li evoluce navrhnout např. kombinační obvod, není nic jednoduššího, než nahradit tabulku kombinačním obvodem, který dává pro všechny vstupní kombinace výsledky shodné s reakcí hledaného obvodu.

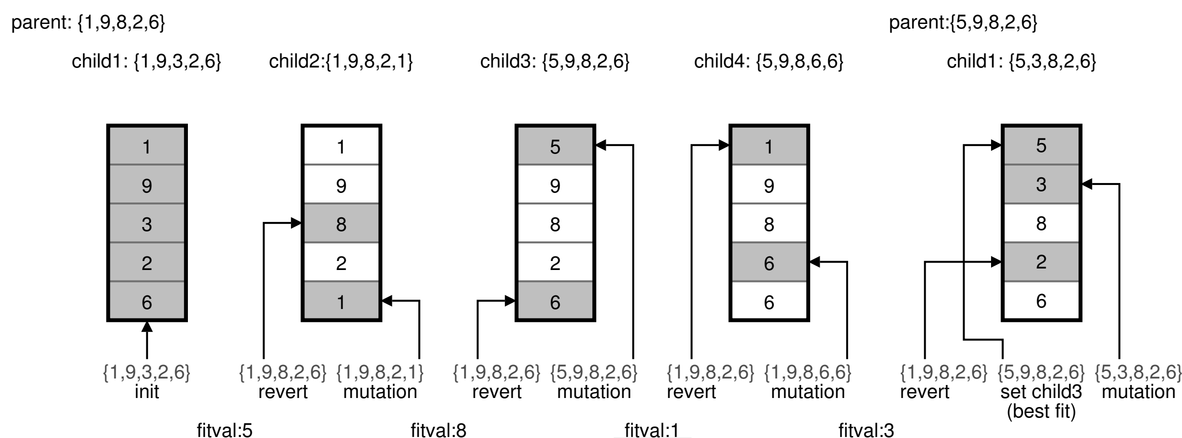
10 Použitý evoluční algoritmus

Jelikož se způsob využití procesoru PowerPC v FPGA liší od klasického softwarového řešení, kde procesor řídí celý evoluční proces, je nutné navrhnout jiný výhodnější evoluční algoritmus. Musíme si uvědomit, že procesor sice získá fitness hodnotu až po ohodnocení banku, ale nový konfigurační řetězec vytváří do stejného banku. Mezitím dochází k ohodnocení sousedního banku a nelze proto použít klasický softwarový sekvenční přístup.

10.1 Horolezecký algoritmus

Nejprimitivnějším algoritmem, který lze velmi snadno implementovat je horolezecký algoritmus. Horolezecký algoritmus je gradientní algoritmus sledující směr největšího růstu. Jelikož se jedná o klasický prohledávací algoritmus, který není populačně orientován, můžeme velmi snadno zařídit nezávislost konfiguračních řetězců umístěných v jednotlivých bancích. Nezávislosti docílíme tehdy, budou-li jednotlivé banky využity pro samostatně vyhodnocující se běhy. Každý běh přitom může prohledávat stavový prostor z jiného počátečního stavu.

Algoritmus generující nový konfigurační řetězec je v tomto případě velmi jednoduchý. Zvolíme k jedinců, kteří budou tvořit okolí a postupně k po sobě následujících kroků generujeme z výchozího řetězce pomocí operátoru mutace nové konfigurační řetězce. Po každém ohodnocení vrátíme provedenou změnu (zmutované slovo) zpět a zapíšeme změnu novou. V případě, že dovolíme pouze jednu mutaci, postačí na získání nového řetězce dva zápisy do paměti. Po k krocích vybereme nejlepšího jedince, který bude tvořit nový počáteční vektor a zahájíme další cyklus.



Obrázek 50: Horolezecký algoritmus, obsah jednoho banku. Zvýrazněné buňky představují zápis hodnoty. Z rodiče $\{1, 9, 8, 2, 6\}$ se generují pomocí jedné mutace čtyři potomci.

Celý cyklus tvorby potomků z rodiče ilustruje obrázek 50. V první (inicializační) fázi musíme vygenerovat konfigurační řetězec rodiče, ze kterého vytvoříme pomocí operátoru mutace prvního potomka. Protože paměť neobsahuje žádná data, je zapotřebí nahrát postupně celý konfigurační řetězec. Po ohodnocení je nutné do paměti nahrát konfiguraci druhého potomka.

Ta se však od konfigurace svého rodiče liší maximálně v tolika slovech, kolik povolíme mutací. Z toho důvodu je výhodnější zapsat do paměti pouze slova z konfigurace rodiče v nichž se první potomek liší od svého rodiče a poté zapsat slova v nichž se od svého rodiče liší druhý potomek. Výhodou tohoto přístupu je krátká doba potřebná k vygenerování nového řetězce, neboť není nutné po každém ohodnocení zapisovat do paměti celý konfigurační řetězec. Na obrázku 50 je znázorněn jeden cyklus algoritmu, který generuje z rodiče čtyři potomky.

Mírně odlišná situace nastává po ohodnocení posledního potomka. Opět se vrátíme na konfigurační řetězec rodiče, ale ihned poté zapíšeme změny, které vedly k vytvoření nejlepšího jedince. Tím dostaneme v paměti nového rodiče a můžeme zahájit další cyklus.

Z hlediska doby potřebné k vytvoření nového jedince si tento algoritmus stojí velmi dobře, neboť během celého výpočetního cyklu je zapotřebí maximálně třikrát tolik zápisů do populační paměti, co jsme povolili mutací. Nemělo by se tedy stát, že by fáze generování nového řetězce trvala delší dobu než fáze ohodnocení. Do paměti musíme nahrát celý konfigurační řetězec pouze při inicializaci.

10.2 Populačně orientovaný algoritmus

Budeme-li chtít vytvořit evoluční algoritmus, je nutné pracovat s populací jedinců. K získání nejjednoduššího evolučního algoritmu vedou následující kroky. Z počáteční populace rodičů, kterou tvoří p jedinců, musíme pomocí operátoru mutace vyrobit a ohodnotit $p \cdot k$ mutantů. Ze všech $p + p \cdot k$ ohodnocených řetězců je zapotřebí vybrat p nejlepších jedinců, kteří budou tvořit novou generaci rodičů.

I zde můžeme, podobně jako u horolezeckého algoritmu, použít p nezávislých banků. Každému banku přiřadíme jednoho jedince z populace rodičů a využijeme jej pro ohodnocení k potomků, kteří vzniknou mutací rodiče. Rozdíl spočívá v ohodnocení poslední p-tice, kdy bychom už měli plnit jednotlivé banky konfiguračními řetězci nové populace. Bohužel však zatím ale nevíme, které řetězce budou novou populaci tvořit. Uveďme si dvě možnosti, jak lze tuto situaci řešit.

1) U prvních $p - 1$ banků negenerujeme žádné nové jedince do paměti a nenastavujeme bit indikující platnost, pouze průběžně řadíme prvních p nejlepších jedinců. Po ohodnocení posledního banku nakonfigurujeme podle nejlepších p jedinců všechny banky. Nevýhoda tohoto přístupu spočívá v tom, že ztrácíme několik taktů, neboť není možné ohodnocovat žádný bank. Výhodou je jednoduchost. Ve většině případů není nutné zapisovat kompletně všechny konfigurační řetězce, neboť s velmi velkou pravděpodobností mohou jedinci, kteří budou tvořit novou populaci, pocházet každý z jiného rodiče. V nejhorším případě bude potřeba přehrát novým konfiguračním řetězcem $p-1$ bank. Tento případ nastane tehdy, pokud všech p nejlepších jedinců vzniklo mutací z jednoho rodiče.

2) Po ohodnocení prvního banku víme, že do konce zbývá $p - 1$ ohodnocení, tudíž konfigurační řetězec, který je nyní nejlépe ohodnocen, musí být určitě součástí nové populace. V nejhorším případě se může dostat z prvního místa až na poslední p-té místo. Můžeme tedy ihned zahájit jeho zápis. Narozdíl od předchozího řešení však nelze zápisy zjednodušit a je zapotřebí zapsat celý konfigurační řetězec. Je totiž velmi málo pravděpodobné, že doposud první nejlépe ohodnocená konfigurace vznikla mutací rodiče z akutálně vyhodnoceného banku. Dá se říci, že za určitého předpokladu platí, že čím více banků, tím více má procesor času na vý-

počet nové konfigurace, protože po přerušení se plní bank, který se bude ohodnocovat tehdy, až se ohodnotí ostatní banky.

Protože potřebujeme získat prvních p nejlépe ohodnocených konfiguračních řetězců, je nutné implementovat vhodný řadicí algoritmus. Vzhledem k tomu, že nás nezajímá, v jaké relaci je ostatních $p \cdot k$ řetězců, je řazení všech $p + p \cdot k$ konfigurací neefektivní. Mnohem výhodnější je řadit položky průběžně.

K realizaci je možné použít lineární pole nebo lineární seznam. Každá zařazená položka musí obsahovat identifikaci řetězce a příslušnou fitness hodnotu. V případě použití lineárního pole se zařazení nového prvku skládá z následujících dvou kroků. Nejprve je nutné vyhledat pozici, na kterou bude nový prvek umístěn. V případě, že použijeme binární vyhledávání, potřebuje tento krok maximálně $\log p$ čtení. Poté je zapotřebí vložit prvek na nalezené místo. Protože je použito pole, je nutné před vložením posunout všechny položky vpravo. Posunutí vyžaduje maximálně $2(p-1)$ čtení a $2(p-1)$ zápisů. Před ohodnocením nové generace, je nutné inicializovat pole záznamy o nových rodičích. K tomu stačí provést pouze aktualizaci identifikace (p zápisů), neboť pole jinak obsahuje potřebné informace z minulé generace. Na jednu generaci připadá celkem $p + k(\log p + 4(p-1))$ paměťových operací, což pro $p = 8$ a $k = 4$ činí 132 operací.

Druhou možností je implementovat algoritmus pomocí lineárního seznamu. Zařazení se opět skládá ze stejných kroků. Protože používáme seznam, musíme nyní pozici vyhledat sekvenčně, což potřebuje maximálně $2p$ čtení (fitness hodnota, ukazatel). Při vkládání nového záznamu stačí přepsat ukazatel předchozího a přidat záznam, tzn. 4 paměťové operace (zápis hodnoty, identifikace a změna dvou ukazatelů). Narozíl od předchozího řešení je však zapotřebí po ohodnocení všech jedinců seskupit po paměti rozptýlené položky. Na tuto operaci potřebujeme přečíst ukazatel a fitness hodnotu a zapsat identifikaci, fitness hodnotu a ukazatel – celkem $5p$ paměťových přístupů. Na jednu generaci připadá celkem $5p + k(4 + 2p)$ paměťových přístupů, což pro $p = 8$ a $k = 4$ činí 120 přístupů.

Vzhledem k velmi podobným výsledkům jsou z hlediska přístupu do paměti obě alternativy téměř ekvivalentní.

10.3 Paralelní algoritmus

Na závěr této kapitoly se zmiňme o možnosti realizace paralelního genetického algoritmu. Stejně jako tomu bylo v předchozích případech, je možné algoritmus implementovat na úrovni software bez nutnosti změny hardware. Paralelní genetický algoritmus se od předchozí varianty liší v tom, že banky rozdělíme na několik nezávislých skupin, nad kterými poběží klasický genetický algoritmus. Jednou za čas se mezi jednotlivými skupinami provede výměna nejlepších jedinců. Výhodou tohoto přístupu je poměrně rychlá konvergence a menší pravděpodobnost uváznutí v lokálním extrému.

11 Evoluční návrh obrazových filtrů

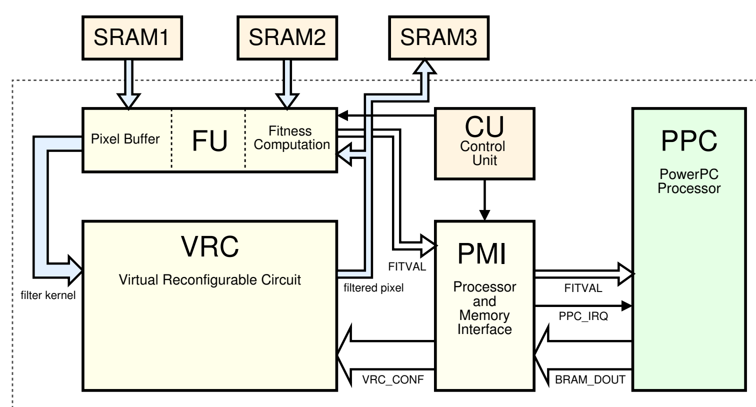
Cílem této kapitoly je ukázát praktickou aplikaci navržené evoluční platformy na problému evolučního návrhu obrazových filtrů a porovnat dosažené výsledky s existujícím hardwarovým řešením. Úkolem evolučního algoritmu je na základě poškozeného vstupního obrazu a požadovaného výstupního obrazu navrhnout obrazový operátor pracující nad filtrovacím oknem 3×3 bodů, který co nejvíce vyhovuje dané specifikaci. Vstupní obraz ve stupních šedi je kódován pomocí osmibitových hodnot.

11.1 Platforma

Abychom mohli řešit úlohu evolučního návrhu obrazových filtrů, je zapotřebí vybavit fitness jednotku pamětí, která bude uchovávat trénovací obrazy. Protože karta Combo6X obsahuje tři externí statické paměti SRAM, můžeme do jedné paměti uložit vstupní obraz, do druhé požadovaný obraz a do třetí paměti získaný vyfiltrovaný obraz. Toto rozdělení dovoluje paralelní přístup k jednotlivým pamětem, což je velmi důležité z hlediska překrytí zápisu nové hodnoty a čtení odezvy rekonfigurovatelného obvodu.

Fitness jednotku, která odpovídá fitness jednotce uvedené v [20], tvoří dvě nezávislé části. První část má za úkol generovat na vstupu rekonfigurovatelného obvodu matici 3×3 osmibitových hodnot, která se získá postupným posuvem filtrovacího jádra po trénovacím vzoru. Protože propustnost pamětí SRAM nedovolí přečíst v jednom taktu všech devět osmibitových hodnot, bylo nutné implementovat tři řádkové buffery FIFO, které musí uchovávat předchozí, následující a aktuálně zpracovávaný řádek vstupního obrazu. Smyslem zavedení bufferů je umožnit za jeden takt přenést hodnoty filtrovacího jádra na vstup VRC obvodu.

Druhá část fitness jednotky má na starosti výpočet fitness hodnoty. Ta se získá jako součet absolutních odchylek mezi požadovanou a vypočtenou hodnotou. Maximální hodnota fitness je závislá na rozměru obrazu $fit_{max} = 255 \cdot m \cdot n$. Pro trénovací obrazy 256×256 bodů tedy na vyjádření fitness hodnoty stačí 24 bitů. Architektura evoluční platformy dovolující návrh obrazových operátorů je uvedena na obrázku 51.



Obrázek 51: Blokový diagram architektury dovolující evoluční návrh obrazových filtrů

Obrazový operátor je reprezentován rekonfigurovatelným obvodem *VRC*, jehož strukturu

jsme diskutovali v kapitole 5.3.3. Datová šířka programovatelných elementů je osm bitů, tzn. vstup i výstup je osmibitový. V tabulce 3 je uveden seznam funkcí, které může programovatelný element realizovat. Jednotlivé funkce byly navrženy experimentálně, detailněji se touto problematikou zabývá [18].

index	funkce	význam	index	funkce	význam
0	255	max. hodnota	8	$x \gg 1$	celočíselné dělení 2
1	x	identita	9	$x \gg 2$	celočíselné dělení 4
2	$255 - x$	inverze	A	$(x \ll 4) \vee (y \gg 4)$	
3	$x \vee y$	bitový součet	B	$x + y$	součet
4	$(255 - x) \vee y$	bitový součet s inverzí	C	$x +^S y$	součet se saturací
5	$x \wedge y$	bitový součin	D	$(x + y) \gg 1$	celočíselný průměr
6	$255 - (x \wedge y)$	inverze bitového součinu	E	$max(x, y)$	maximum
7	$x \oplus y$	exkluzivní bitový součet	F	$min(x, y)$	minimum

Tabulka 3: Seznam funkcí, které může realizovat programovatelný element VRC.

Protože je fáze přípravy nového konfiguračního řetězce a vlastní konfigurace překryta s fází evaluace, určuje dobu potřebnou k ohodnocení jednoho kandidátního řešení pouze doba výpočtu fitness hodnoty t_{eval}, kterou lze získat následovně.

$$t_{eval} = (m-2)(n-2)\frac{1}{f} = (m-2)(n-2)\frac{1}{50}\mu s,$$

kde $m \times n$ je rozměr obrazu v bodech a f je taktovací frekvence fitness jednotky a rekonfigurovatelného obvodu (50 MHz). Doba potřebná k ohodnocení jednoho kandidátního řešení je pro názornější představu uvedena v tabulce 4. Mimo to je uveden i počet filtrů, který je možné během jedné sekundy ohodnotit.

rozměr obrazu	doba ohodnocení	ohodnocení za sekundu
32×32	18 μs	55 555
64×64	77 μs	13 007
128×128	318 μs	3 149
256×256	1 291 μs	775

Tabulka 4: Doba potřebná k ohodnocení jednoho kandidátního řešení a počet ohodnocených řešení za sekundu v závislosti na rozměru trénovacího obrazu.

11.2 Výsledek syntézy

Rekonfigurovatelná platforma je popsána pomocí jazyka VHDL, simulována s využitím simulačního nástroje ModelSim a syntetizována systémem Precision a Xilinx ISE. Protože simulátor ModelSim je schopen po přidání knihoven simulovat i chování procesoru PowerPC, bylo možné ověřit funkčnost navrženého systému jako celku.

Výsledek syntézy pro FPGA Virtex II Pro 2VP50ff1517 je shrnut v tabulce 5. V tabulce je uveden i výsledek syntézy pro systém bez virtuálního rekonfigurovatelného obvodu (VRC byl nahrazen jednoduchým obvodem), aby bylo možné získat představu o počtu zdrojů, které VRC zabírá.

VRC	IO bloky	BRAM	CLB	DFF
dostupno	852	232	23 616	49 788
4 × 8 využití	602 70%	12 5%	4 591 20%	3 638 7%
bez VRC využití	602 70%	12 5%	1 240 5%	2 479 5%

Tabulka 5: Výsledek syntézy pro Virtex II Pro 2VP50ff1517

11.3 Experimentální vyhodnocení

Cílem této kapitoly bude porovnat řešení využívající procesor PowerPC s původním hardwarovým řešením publikovaným v [20]. Protože evoluční platforma využívající procesor PowerPC umožňuje implementaci libovolného genetického algoritmu, bude úkolem zjistit, zda má sofistikovanější genetický algoritmus vliv na kvalitu získaných řešení. Další otázkou, na kterou se budeme snažit nalézt odpověď, je, jaký vliv má na kvalitu získaných řešení generátor pseudonáhodných čísel.

Aby bylo možné provést mnoho experimentů, byly použity osmibitové trénovacími obrazy o rozměru 128 × 128 bodů. Jedná se o kompromis mezi dobou potřebnou pro ohodnocení kandidátního řešení a množstvím informace, které obraz obsahuje. Úkolem evoluce bylo do předem stanoveného počtu evaluací nalézt řešení. Horní limit byl stanoven na 49152 evaluací, což odpovídá přibližně 15 sekundám výpočtu. Upřesňující informace budou uvedeny u jednotlivých experimentů.

Získané výsledky budeme pro jednoduchost prezentovat na úloze filtrace výstřelového šumu a hledání Sobelova operátoru [17]. Tyto úlohy byly vybrány z toho důvodu, že se z hlediska evoluce jedná o různě složité problémy. Zatímco evoluční návrh filtru odstraňujícího výstřelový šum se jeví jako poměrně snadná úloha, ukazuje se evoluční návrh Sobelova operátoru jako úloha obtížnější, která vyžaduje větší množství evaluací.

Na závěr ukážeme několik získaných výsledků z ostatních experimentů. Kromě úloh uvedených v následujících odstavcích bylo zopakováno i několik experimentů uvedených v [20].

11.3.1 Evoluční algoritmus

Abychom získali objektivní výsledky, které nejsou závislé na konkrétní volbě parametrů, byly následující experimenty provedeny přes různé hodnoty četnosti mutace a pro oba typy generátorů pseudonáhodných čísel (viz dále). Celkem bylo provedeno kolem 2000 experimentů. Úkolem bude porovnat následující tři algoritmy, které byly implementovány v procesoru PowerPC.

1. algoritmus náhodného prohledávání (RS), který odpovídá algoritmu použitému v HW genetické jednotce. Defacto se jedná o horolezecký algoritmus, jehož okolí je tvořeno pouze jedním jedincem. Pomocí mutace se ze stávající konfigurace vytvoří nový potomek, který se ohodnotí. Pokud je získaný potomek ohodnocen lépe, než byl ohodnocen jeho rodič, nahradí potomek svého rodiče.

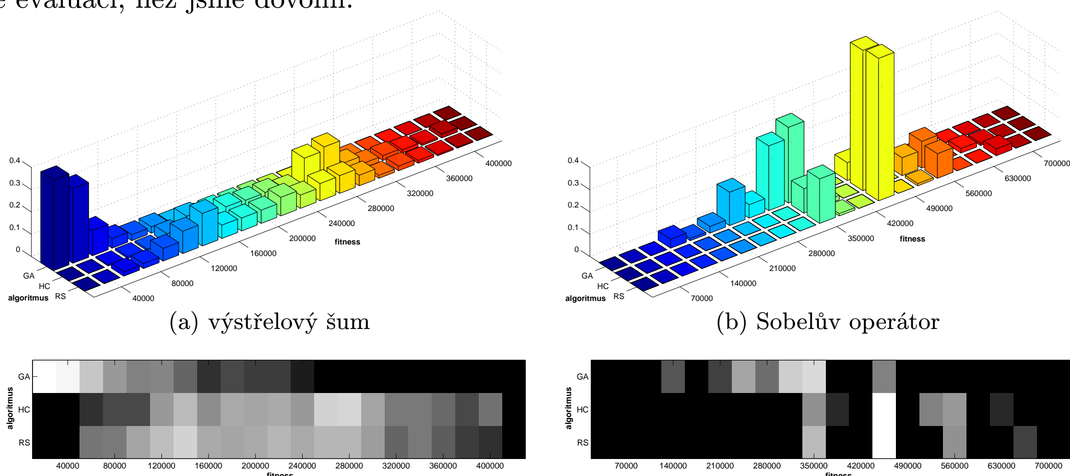
2. horolezecký algoritmus (HC), který odpovídá popisu uvedeném v kapitole 10.1. V tomto algoritmu bylo použito okolí tvořené osmi jedinci.

3. genetický algoritmus (GA), diskutovaný v kapitole 10.2. Parametry genetického algoritmu byly zvoleny následovně. Velikost populace byla stanovena na osm jedinců. Z každého jedince je pomocí operátoru mutace generováno osm nových potomků. Novou populaci tvoří nejlepších osm jedinců z 64 potomků a původních 8 rodičů.

algoritmus / fitness hodnota	minimum	maximum	průměr	odchylka
výstřelový šum				
genetický algoritmus (GA)	8 312	307 024	36 706	39 887
horolezecký algoritmus (HC)	54 201	2 643 688	264 206	153 484
náhodné prohledávání (RS)	46 108	559 296	207 438	101 931
Sobelův operátor				
genetický algoritmus (GA)	118 001	453 609	298 995	70 494
horolezecký algoritmus (HC)	334 438	1 354 168	469 226	108 260
náhodné prohledávání (RS)	315 272	632 796	445 193	68 242

Tabulka 6: Vliv použitého algoritmu na kvalitu nalezeného řešení. Ukázka nejlepší, nejhorší a průměrné fitness hodnoty.

Získané výsledky jsou shrnuty v tabulce 6. Kvalitu algoritmu je možné určit na základě průměrné fitness hodnoty a její standartní odchylky. V případě genetického algoritmu jsou oproti ostatním algoritmům oba ukazatele výrazně menší. U Sobelova operátoru není rozdíl tak markantní jako u výstřelového šumu, nicméně i zde můžeme sledovat značné zlepšení. Menší rozdíl může být způsoben mnohem větší obtížností řešeného problému, který potřebuje více evaluací, než jsme dovolili.

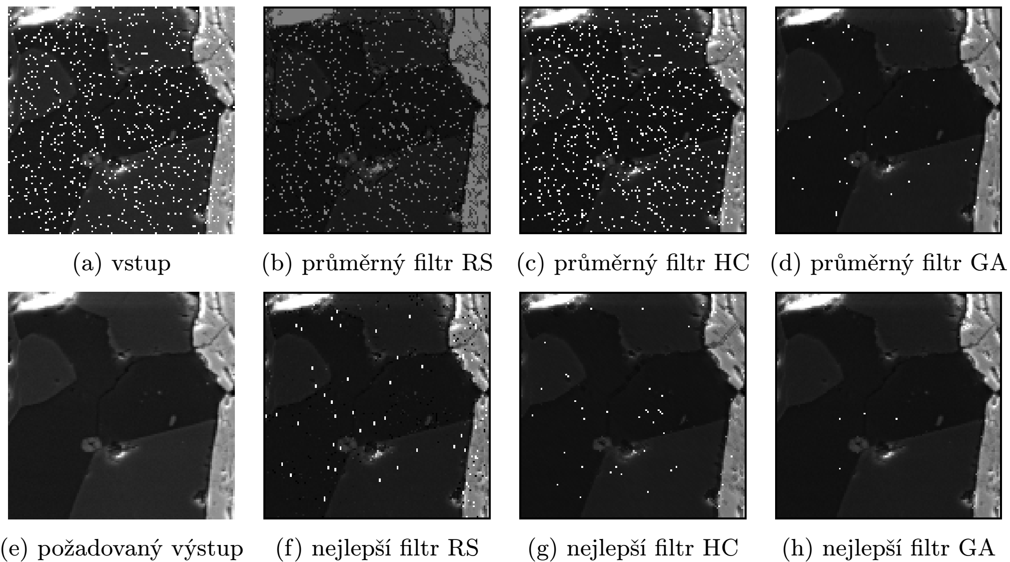


(a) výstřelový šum

(b) Sobelův operátor

Obrázek 52: Histogramy rozložení fitness hodnot v závislosti na použitém algoritmu

Pro názornost je na obrázku 52 uveden trojrozměrný a odpovídající dvojrozměrný histogram, který zachycuje rozložení fitness hodnot nalezených řešení v závislosti na použitém algoritmu. Histogram je normován počtem experimentů, součet přes všechny hodnoty je tedy roven 1. V histogramu je možné opět sledovat značný vliv genetického algoritmu, což dokazuje koncentrace nalezených řešení kolem nižších fitness hodnot. I v případě Sobelova operátoru vidíme zvýšení četnosti výskytu u nižších fitness hodnot.

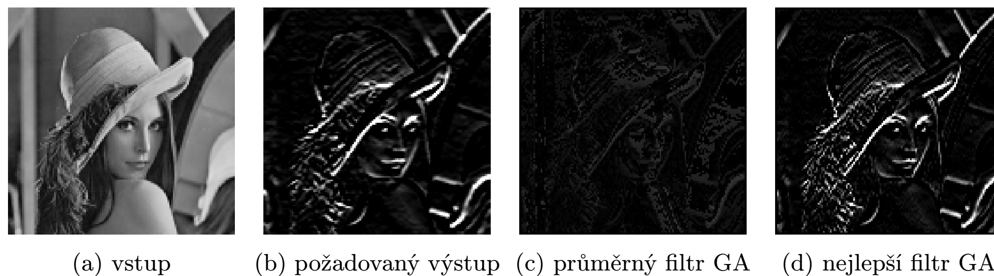


(a) vstup (b) průměrný filtr RS (c) průměrný filtr HC (d) průměrný filtr GA

(e) požadovaný výstup (f) nejlepší filtr RS (g) nejlepší filtr HC (h) nejlepší filtr GA

Obrázek 53: Ukázka odezvy filtru s průměrnou fitness hodnotou (nahoře) a nejlepšího nalezeného filtru (dole) při použití různých algoritmů

Na obrázku 53 jsou znázorněny odezvy filtru s průměrnou a minimální fitness hodnotou, které korespondují s hodnotami uvedenými v tabulce 6. Odezva filtru s průměrnou fitness hodnotou nám dává informaci o tom, jaké filtry je nejpravděpodobněji možné získat za 15 sekund evoluce. Jelikož je při použití genetického algoritmu i průměrný filtr velmi kvalitní, ukazuje se jako zajímavá myšlenka využít vyvíjejícího se filtru v praxi. Vhodnou aplikací by např. mohlo být adaptivní zpracování obrazů získaných z kamery, která snímá výrobní linku s relativně stálým prostředím a pro toto prostředí typickým šumem.



(a) vstup (b) požadovaný výstup (c) průměrný filtr GA (d) nejlepší filtr GA

Obrázek 54: Ukázka odezvy filtru s průměrnou fitness hodnotou a nejlepšího nalezeného filtru při použití genetického algoritmu

Obrázek 54 znázorňuje opět odezvu filtru s průměrnou a minimální fitness hodnotou, ovšem pro úlohu návrhu Sobelova operátoru. Operátor s průměrnou fitness hodnotou je však v tomto případě prakticky nepoužitelný. Abychom získali kvalitnější výsledek, bylo by nutné zvýšit počet evaluací.

11.3.2 Generátor náhodných čísel

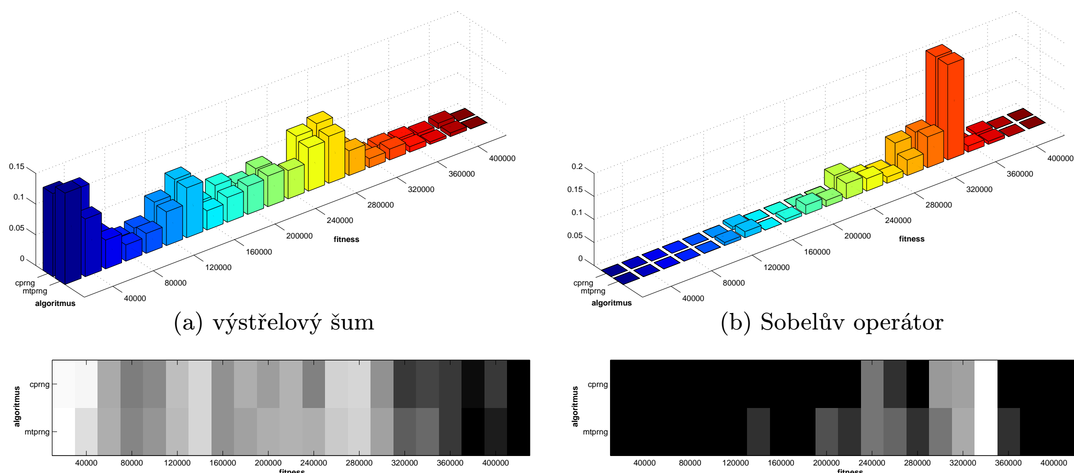
Evoluční algoritmus je stochastický prohledávací algoritmus řízený pravděpodobností, avšak hodnoty generované běžnými generátory pseudonáhodných čísel (PRNG) nejsou čistě náhodné. Cílem tohoto experimentu je zjistit vliv kvality generátoru pseudonáhodných čísel na kvalitu získaných řešení. V PowerPC byl implementován běžný kongruentní generátor a mnohem sofistikovanější generátor Mersenne Twister (MT) publikovaný v [10].

Generátor MT se vyznačuje vysokou výkonností (používá jednoduché 32bitové logické operace a posuvy), poměrně jednoduchým algoritmem a oproti klasickému generátoru vyšší kvalitou generovaných čísel. Tento generátor byl vybrán především z důvodu nízkého počtu operací, které je nutné provést k získání nové náhodné hodnoty. V čistě hardwarové realizaci [20] byl generátor realizován pomocí LFSR (Linear Feedback Shift Register) registru, který patří, podobně jako MT, do kategorie kvalitnějších generátorů [34].

Parametry kongruentního generátoru pracujícího na 32bitech odpovídají běžně používaným parametrům (např. standartní knihovna C). Nová pseudonáhodná hodnota rnd_{i+1} se pomocí staré hodnoty vypočte následovně

$$rnd_{i+1} = 1103515245\ rnd_i + 12345$$

Celkem bylo provedeno 2000 běhů evoluce pro výstřelový šum a 200 běhů pro Sobelův operátor. Experimenty probíhaly přes různé hodnoty četnosti mutace a přes všechny algoritmy uvedené v předchozí kapitole.



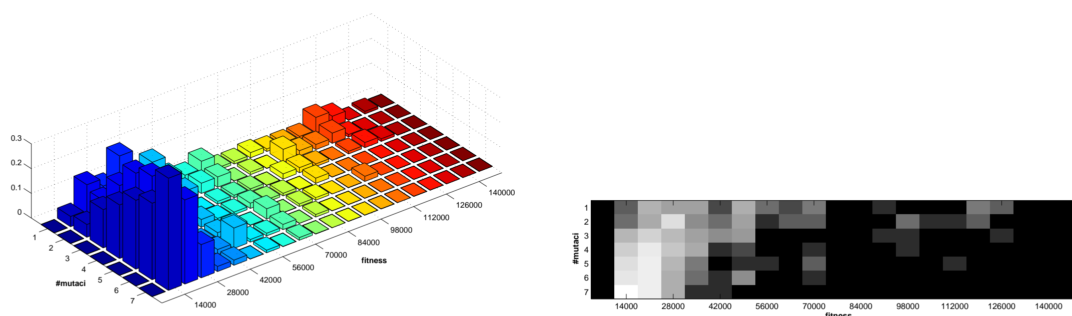
(a) výstřelový šum (b) Sobelův operátor

Obrázek 55: Srovnání kongruentního (cprng) a mersenne twister (mtprng) PRNG generátoru při řešení a) impulsího šumu a b) Sobelova operátoru.

Získané histogramy jsou uvedeny na obrázku 55. Porovnáme-li střední hodnoty a odchylky, dojdeme k závěru, že oba implementované generátory dosahují stejných výsledků. Pro kongruentní generátor vychází průměrná fitness hodnota 168 834 a odchylka 152 501, pro Mersenne Twister 167 552 a odchylka 137 778. Pro evoluční návrh obrazových filtrů lze používat jednoduchý kongruentní generátor, navíc nová pseudonáhodná hodnota může být s využitím výkonné MAC instrukce vypočtena v několika málo taktech.

11.3.3 Četnost mutace

Podobně jako v [20] byl proveden experiment, jehož cílem je ukázat, že i při použití genetického algoritmu vede zvyšování četnosti mutace k zlepšení průměrné fitness hodnoty a je možné pro řešený problém stanovit jeho optimální hodnotu. Četnost mutace je parametr, který udává maximální počet bitů konfiguračního řetězce, který může změnit svou hodnotu na opačnou po aplikaci operátoru mutace. Jelikož snahou není nalézt optimální hodnotu tohoto parametru, ale pouze ukázat, že tento efekt existuje, byl parametr četnosti volen v rozsahu $1 - 7$. Pro každou hodnotu bylo provedeno 100 experimentů a jako trénovací obraz byl použit obraz Leny o rozměru 128×128 bodů znehodnocený výstřelovým šumem.



Obrázek 56: Histogram rozložení fitness hodnot pro různé hodnoty parametru četnosti mutace

Získané výsledky jsou shrnuty v tabulce 7. Z poklesu průměrné fitness hodnoty a standartní odchylky můžeme pozorovat, že zvyšování četnosti mutace má pozitivní vliv na kvalitu nalezených řešení. Tento efekt můžeme sledovat i v histogramu uvedeném na obrázku 56. Se vzrůstající četností mutace se nalezená řešení více koncentrují kolem nižších fitness hodnot.

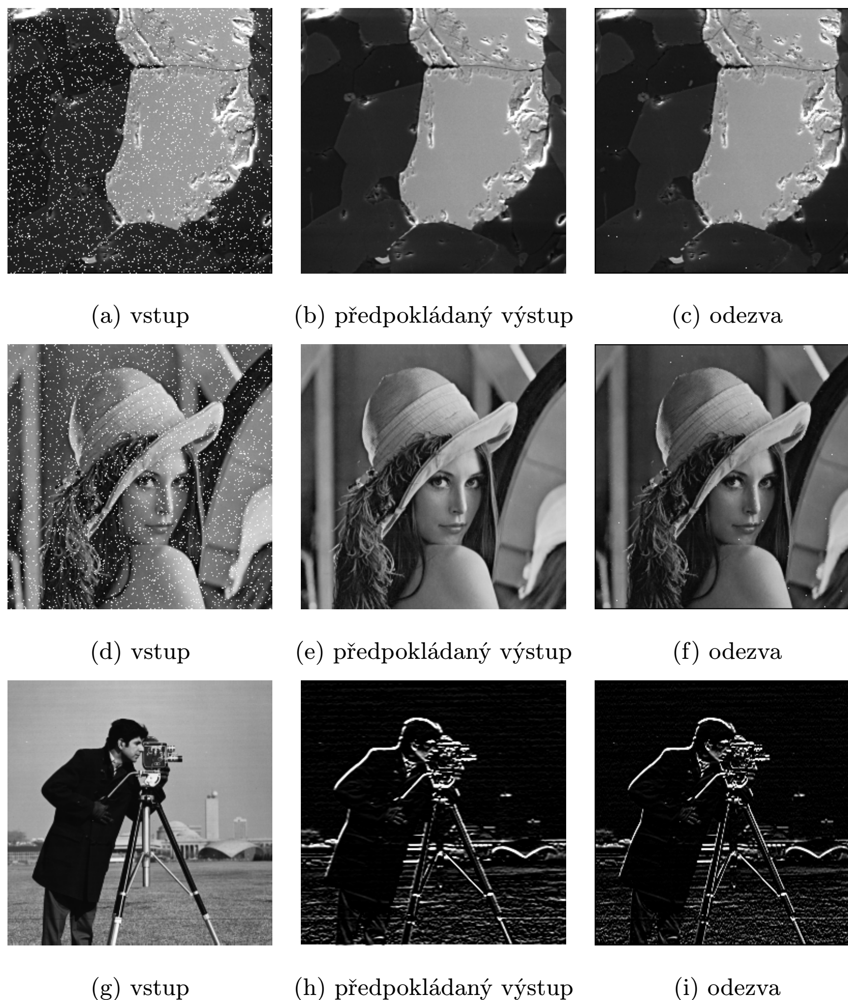
četnost mutace	minimum	maximum	průměr	odchylka
1	9 490	289 442	66 018	60 303
2	9 976	307 024	50 408	46 175
3	9 149	175 148	38 552	36 514
4	9 120	214 674	27 543	25 511
5	9 284	192 914	28 065	30 826
6	9 685	192 573	24 473	22 452
7	8 312	152 775	18 624	16 280

Tabulka 7: Vliv četnosti mutace na kvalitu nalezeného rešení. Ukázka nejlepší, nejhorší a průměrné fitness hodnoty.

11.3.4 Generalizace

Ačkoliv není evoluční algoritmus schopen zaručit obecnost, podařilo se navrhnout operátory, které dávají vynikající výsledky i pro jiné obrazy, než na které byly trénovány. Abychom zaručili obecnost, museli bychom předložit odezvu na všech $256^9 = 2^{72}$ možností, kterých může nabývat filtrovací jádro 3×3 bodů. Protože se však tyto filtry podařilo nalézt i za pomocí obrazů 128×128 bodů, které obsahují maximálně 15876 různých kombinací, poskytují

zřejmě pro evoluční návrh dostatečné množství informace. Pomocí evoluce byl navržen filtr pro výstřelový šum a Sobelův operátor, oba pracující nezávisle na trénovacím obraze.



(a) vstup (b) předpokládaný výstup (c) odezva

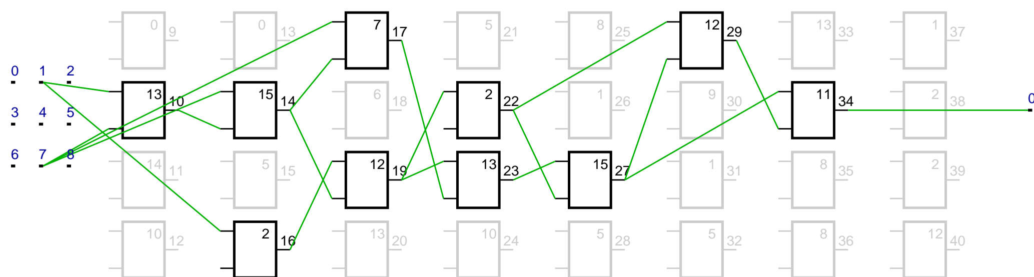
(d) vstup (e) předpokládaný výstup (f) odezva

(g) vstup (h) předpokládaný výstup (i) odezva

Obrázek 57: Odezvy filtrů na obrazy 256×256 trénovaných na odlišné obrazy 128×128

Na obrázku 57 jsou uvedeny odezvy nalezených operátorů, které byly trénovány pomocí obrazů 128×128 bodů uvedených na obrázku 53 a 54. Porovnáme-li odezvu filtru výstřelového šumu (obrázek 57.c) s obrazem získaným pomocí median filtru (obrázek 21.c), zjistíme, že nalezený filtr zachovává oproti medianu mnohem více detailů. Podobně odezva nalezeného Sobelova operátoru (obrázek 57.i) dosahuje oproti konvenčnímu operátoru (obrázek 57.h) mnohem vyšší ostrosti.

Ostřejší odezva je typická pro nelineární filtry. Problémem třídy nelineárních filtrů je neexistence konvenčních metod umožňujících jejich návrh. Ukazuje se, že tento problém lze překonat, využijeme-li evolučního návrhu. Evoluce má možnost prozkoumat mnohem větší prostor možných řešení a nalézt operátory, které neumíme pomocí nám dostupných metod získat.



Obrázek 58: Schéma evolucí navrženého Sobelova operátoru. Fukce odpovídají funkcím uvedeným v tabulce 3.

Přepis operátoru nalezeného pomocí evoluce, jehož schéma je uvedeno na obrázku 58, do jazyka C. Vstupem funkce *filter()* je devítice osmibitových hodnot filtrovacího jádra, výstupem je osmibitová hodnota. Protože jsou použity poměrně jednoduché funkce, je možné nalezený filtr velmi efektivně implementovat v hardware.

```
uint8 filter(uint8 kernel[9]) {
  uint i14,i17,i19,i22,i27,i29;

  i14 = min((kernel[1] + kernel[7]) >> 1, kernel[7]);
  i17 = i14 ^ kernel[7];
  i19 = min(i14 + (255 - kernel[1]), 255);
  i22 = 255 - i19;
  i27 = min(i22, (i17 + i19) >> 1);
  i29 = min(i22 + i27, 255);

  return (i27 + i29) & 0xff;
}
```

Implementace konvenčního Sobelova operátoru.

```
uint8 filter(uint8 kernel[9]) {
  int i;

  i = kernel[0] + 2*kernel[1] + kernel[2];
  i = i - (kernel[6] + 2*kernel[7] + kernel[8]);
  i = max(i, 0);
  i = min(i, 255);

  return i;
}
```

12 Závěr

V této práci jsme se seznámili s problematikou vyvíjejících se obvodů a navrhli generickou rekonfigurovatelnou platformu, která umožňuje akceleraci úloh evolučního návrhu. Jedná se o kompletně hardwarové řešení, celý systém je implementován uvnitř rekonfigurovatelného hradlového pole FPGA a využívá zabudovaný procesor PowerPC. Abychom dosáhli maximálního výkonu, je časově náročnější část evolučního algoritmu (ohodnocení a výpočet fitness hodnoty) realizována v hardware a výpočetně náročnější část (tvorbu nových konfiguračních řetězců) má na starosti procesor PowerPC. Jelikož odpadla nutnost provádět pomalou softwarovou simulaci kandidátního obvodu, dosahuje se až padesátinásobného zrychlení oproti běžnému softwarovému řešení. Pomocí této platformy je možné řešit řadu odlišných úloh, jedná se např. o evoluční návrh obrazových filtrů, návrh kombinačních obvodů, řadicích sítí apod. Změna úlohy vyžaduje pouze změnu struktury rekonfigurovatelného obvodu a případně fitness jednotky starající se o výpočet fitness hodnoty. Získaný systém nám dává díky spojení software a hardware mnoho nových příležitostí, můžeme sledovat vliv evolučního algoritmu na kvalitu nalezených řešení, hledat vhodné parametry, vhodný algoritmus apod. Výhodou je nejen možnost velmi snadno měnit evoluční algoritmus, ale především během relativně krátké doby získat velké množství výsledků.

Funkčnost byla demonstrována na evolučním návrhu obrazových operátorů. Ukázali jsme, že navržená platforma dosahuje oproti kompletně hardwarovému řešení publikovaném v [20] mnohem lepších výsledků. Návrhem vhodného evolučního algoritmu jsme dosáhli dramatického zvýšení kvality nalezených filtrů při zachování stejné výkonnosti. Díky zmenšení průměrného počtu potřebných ohodnocení je možné uvažovat o využití vyvíjejících se obvodů v adaptivních systémech.

Pozitivním výsledkem je objevení obrazových operátorů, které vykazují vysokou kvalitu i pro obrazy, pro které nebyly trénovány. Dále se podařilo nalézt operátor pro detekci hran, který dává mnohem ostřejší výsledek než běžný sobelův operátor. Navíc vzhledem k použitým funkcím lze nalezené operátory velmi snadno implementovat v hardware a získat kvalitní filtry s velmi vysokou propustností. Výhodou evolučního návrhu je možnost navrhovat obvody, které se skládají z námi definovaných (téměř libovolných) funkcí. Pomocí evoluce lze nalézt řešení, která není schopen pomocí běžných návrhových metod konstruktér získat.

Abychom dosáhli většího výkonu, mohli bychom se pokusit více zoptimalizovat fitness jednotku a virtuální rekonfigurovatelný obvod tak, aby pracovali na taktovacím kmitočtu 100 MHz. Touto úpravou bychom dosáhli dvojnásobného počtu ohodnocených operátorů za stejnou dobu. Z hlediska výsledků je zajímavou myšlenkou přejít na operátory pracující nad filtrovacím oknem 5×5, případně 9×9 obrazových bodů. Tato změna by mohla přinést řadu překvapivých výsledků. Jelikož se při evolučním návrhu kombinačních obvodů ukazuje, že hodnota parametru l-back má značný vliv na úspěšnost nalezení řešení, bylo by vhodné modifikovat virtuální rekonfigurovatelný obvod tak, aby umožňoval alespoň propojení o dva sloupce zpět.

Reference

[1] Evans, J.: An efficient FIR filter architecture. In: Int. Symposium. Acoust., Speach, Signal Processing, Vol. 1, str. 627-630

[2] Gwaltney, D., Dutton, K.: A VHDL Core for Intrinsic Evolution of Discrete Time Filters with Signal Feedback. In Proc. of 2005 NASA/DoD Conference on Evolvable Hardware, IEEE Comp. Society Press, 2005, str. 43-50

[3] Higuchi, T. et al.: Evolving Hardware with Genetic Learning: A First Step Towards Building a Darwin Machine. In: SAB'92: Proc. of the 2rd International Conference on Simulated Adaptive Behaviour, MIT Press, Cambridge MA 1993, str. 417-424

[4] Higuchi, T. et al.: Evolvable Systems: From Biology to Hardware. First International Conference, ICES 96, LNCS 1259, Springer Verlag, 1997, str. 55-78

[5] Higuchi, T., Kajihara, N.: Evolvable Hardware Chips for Industrial Applications. Communications of the ACM, Vol. 42, No. 4, 1999, str. 66-66

[6] Ifeachor, E. C., Jervis, B. W.: Digital Signal Processing A Practical Approach. Second Edition, New York, Prentice-Hall, 2002

[7] Jiří, J. : Číslicová filtrace, analýza a restaurace signálů. VUTIUM, Brno, 2002

[8] Knuth, D.: The Art of Computer Programming (2nd ed.). Addison Wesley, 1998

[9] Kvasnička, V., Pospíchal, J., Tiňo, P.: Evolučné algoritmy. Vydavateľstvo STU v Bratislave, 2000

[10] Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), 1998.
`http://www.math.keio.ac.jp/matumoto/emt.html` (citováno květen 2006)

[11] Miller, J., Job, D., Vassilev, V.: Principles in the Evolutionary Design of Digital Circuits – Part I. Genetic Programming and Evolvable Machines. Vol. 1, No. 1, 2000, str. 8-35

[12] Miller, J.: Evolution of Digital Filters Using a Gate Array Model. In: Proc. of the Evolutionary Image Analysis, Signal Processing and Telecommunications Workshop. LNCS 1596, Springer-Verlag, 1999, str. 121-132

[13] Miller, J.: Digital Filter Design at Gate-level using Evolutionary Algorithms. In Proc. of the Genetic and Evolutionary Computation Conference (GECCO99). Morgan Kaufmann, 1999, str. 1127-1134

[14] Miller, J. F., Vassilev V.: Scalability Problems of Digital Circuit Evolution. In: A. Lohn et al.: The Second NASA/DoD workshop on Evolvable Hardware, IEEE computer Society, Los Alamitos, 2000

[15] Miller, J., Thomson, P.: Cartesian Genetic Programming. In: Proc. Of the 3rd European Conference on Genetic Programming EuroGP, LNCS 1802, Springer Verlag, Berlin 2000, str. 121-132

[16] Murakawa, M. et al.: Analogue EHW chip for intermediate frequency filters, in Evolvable Systems: From Biology to Hardware. Second Int. Conf., ICES 98, Springer Verlag, 1998, str. 134-143

[17] Ramesh, J.: Machine Vision. Boston, McGraw-Hill, 1995.

[18] Sekanina, L.: Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer Verlag, Berlin, 2003

[19] Sekanina, L., Vašíček, Z.: On the Practical Limits of the Evolutionary Digital Filter Design at the Gate Level. Applications of Evolutionary Computing. LNCS, Springer, 2006

[20] Sekanina, L., Martínek, T.: An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA, In: Lecture Notes in Computer Science, roč. 2005, č. 3637, Berlin, str. 76-85

[21] Shmulevich, I., Melnik, V., Egiazarian, K.: Optimization of Stack Filters Using Sample Selection Probabilities. Proceedings of IEEE-EURASIP Workshop on Nonlinear Signal and Image Processing, Antalya, Turkey, 1999.

[22] Smith, S. W.: Digital Signal Processing. California Technical Publishing, San Diego, 1999

[23] Thompson, A.: Hardware Evolution. Brighton, U.K., University of Sussex, 1996

[24] Thompson, A.: Silicon Evolution. In: GP'96, Proc. of the 2nd Genetic Programming Conference, Stanford, USA, 1996, str. 444-452

[25] Torresen, J.: Evolvable Hardware as a New Computer Architecture. FPL 2004. LNCS, Springer-Verlag, Berlin, 2004

[26] Upegui, A., Sanchez, E.: Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs. ICES 2005. LNCS, Springer Verlag, 2005

[27] Vassilev, V.K., Job, Dominic and Miller, J. F. : Towards the Automatic Design of More Efficient Digital Circuits. In: Proc. 2nd NASA/DOD Workshop on Evolvable Hardware, 2000, str. 151-160.

[28] Vašíček, Z., Sekanina, L.: Evoluční návrh kombinačních obvodů. Elektrorevue 2004/43, `http://www.elektrorevue.cz/clanky/04043/` (citováno prosinec 2005)

[29] Vašíček, Z.: Evolutionary Design of Digital Circuits at the Gate Level. In: Proc. of the 11th Student Conference EEICT, Brno, 2005

[30] Vích, R., Smékal, Z.: Číslicové filtry. Praha, Academia, 2000

[31] Wendt, P. D., Coyle, E.J.: Stack Filters. IEEE Transactions on Acoustics, Speech and Signal Processing. Vol. 34, No. 4, 1986, str. 898-911

[32] Accel Chip, http://www.accelchip.com (citováno květen 2006)

[33] Anadigm Inc., http://www.anadigm.com (citováno květen 2006)

[34] Efficient Shift Registers, LFSR Counters, and Long PseudoRandom Sequence Generators, http://www.xilinx.com/bvdocs/appnotes/xapp052.pdf (citováno květen 2006)

[35] Genobyte Project, http://www.genobyte.com/cbm.html (citováno květen 2006)

[36] IBM Power Architecture Family, http://www-306.ibm.com/ (citováno květen 2006)

[37] Liberouter Project, http://www.liberouter.org/ (citováno květen 2006)

[38] PowerPC 405 Processor Block Reference Guide, http://www.xilinx.com/bvdocs/userguides/ug018.pdf (citováno květen 2006)

[39] PowerPC Processor Reference Guide, http://www.xilinx.com/bvdocs/userguides/ppc_ref_guide.pdf (citováno květen 2006)

[40] Virtex-II Pro and Virtex-II Pro X Platform FPGAs, http://www.xilinx.com/bvdocs/publications/ds083.pdf&e=15055 (citováno květen 2006)

[41] Xilinx Inc., http://www.xilinx.com (citováno květen 2006)