VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

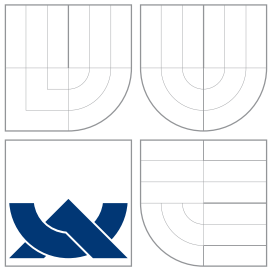
NÁVRH PAMĚTI CACHE PRO SÍŤOVÉ APLIKACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

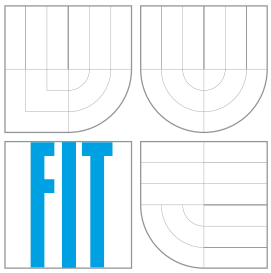
AUTOR PRÁCE
AUTHOR

LUKÁŠ SOĽANKA

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH PAMĚTI CACHE PRO SÍŤOVÉ APLIKACE

CACHE MEMORY DESIGN FOR NETWORK APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ SOĽANKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KOŘENEK
SUPERVISOR

BRNO 2007

Zadání

1. Seznamte se s kartou COMBO6X a technologií Virtex-II Pro od firmy Xilinx.
2. Nastudujte problematiku návrhu pamětí CACHE.
3. Navrhněte architekturu paměti CACHE s ohledem na využití v síťových aplikacích.
4. Proveďte implementaci navržené architektury v jazyce VHDL s ohledem na syntézu do technologie FPGA. Snažte se, aby výsledný kód byl generický a umožňoval podle generických parametrů přizpůsobit paměť CACHE požadavkům širokého spektra síťových aplikací.
5. Vytvořenou implementaci ověřte v simulacích i v hardware na desce COMBO6X.
6. V závěru diskutujte dosažené výsledky. Zaměřte se zejména na použití paměti v různých aplikacích.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Táto práca sa zaoberá návrhom a implementáciou generickej cache pamäte pre široké spektrum sieťových aplikácií. Najprv sú diskutované hlavné aspekty na výkonnosť. V náväznosti na to je navrhnutá architektúra a pamäť implementovaná s rešpektom pre zvolenú cieľovú technológiu. Hlavnými generickými parametrami sú: dátová šírka, veľkosť riadku, asociativita, počet riadkov a spôsob výberu obete. Cache podporuje zreťazené spracovanie, ktoré umožňuje každý hodinový cyklus vykonať jednu požiadavku. Pre overenie funkcionality bola výsledná komponenta dôkladne odsimulovaná a nakoniec bola jej činnosť odskúšaná v hardware na doske Combo6X.

Kľúčové slová

cache, sieť, FPGA

Abstract

This thesis deals with design and implementation of generic cache memory for a wide range of network applications. Firstly, aspects with influence on performance are discussed. Then architecture is proposed and implemented with respect to the given technology. The main selectable parameters of the design are: data path width, line size, associativity, number of lines and replacement policy. Cache is also pipelined and therefore is able to process one request for reading or writing every clock cycle. The resulting component has been thoroughly simulated to verify its functionality and finally, its operation has also been tested in hardware on the Combo6X board.

Keywords

cache, network, FPGA

Citácia

Lukáš Soľanka: Návrh paměti CACHE pro síťové aplikace, bakalářská práce, Brno, FIT VUT v Brně, 2007

Návrh paměti CACHE pro síťové aplikace

Prehlásenie

Vyhlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jána Kořenka. Ďalšie informácie mi poskytli moji kolegovia z projektu Liberouter. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Lukáš Soľanka
14. mája 2007

Poďakovanie

Rád by som poďakoval svojmu vedúcemu Ing. Jánovi Kořenkovi a kolegovi Bc. Martinovi Žádnikovi za poskytnuté rady a informácie. Taktiež ostatní kolegovia z projektu Liberouter mi poskytli veľa cenných informácií a boli nápomocní, za čo im ďakujem.

Obsah

Kapitola 1

Úvod

V dnešnej dobe je možné vo výpočtových systémoch pozorovať stále rastúce rozdiely vo výkonnosti medzi procesorom a hlavnou pamäťou. Vytvorenie pamäte, ktorá by mala čo najväčšiu kapacitu a zároveň by sa svojou rýchlosťou vyrovnala procesoru je veľmi drahé, preto sa do tohto systému vkladá ďalší článok – cache. Cache je malá ale rýchla pamäť, ktorá dočasne obsahuje informácie, ktoré sú práve používané. Prístup k dátam v cache je oveľa kratší, preto procesor strávi oveľa menej času pri čakaní na čítanie inštrukcií alebo dát. V dnešnej dobe prakticky všetky počítačové systémy obsahujú cache pamäte.

Úspech cache je založený na princípe známom ako lokalita odkazov [2]. Odkazy do pamäte, na inštrukcie a aj na operandy, sú počas práce procesora zhlukované. Princíp lokality má dve podoby: časová lokalita, kde referencie sú zhlukované v čase a priestorová lokalita, kde referencie sú zhlukované v (adresovom) priestore. Zatiaľ čo pri vykonávaní programu platia obidve varianty, bolo ukázané, že v sieťovej prevádzke platí iba princíp časovej lokality [11]. To je však stále postačujúca podmienka pre výhodné použitie cache pre rozmanité sieťové aplikácie.

Práve rozmanitosť aplikácií vytvára nutnosť pre každú požiadavku vytvoriť špeciálnu cache. Tento postup, aj keď v mnohých prípadoch jeho výsledkom je lacný a pre konkrétnu potrebu vysoko optimalizovaný systém, zlyháva pri nutnosti všeobecného modelu, prispôsobiteľnému rôzne sa meniacim požiadavkám aplikácií. Táto práca si kladie za cieľ vytvorenie generickej cache pamäte použiteľnej pre široké spektrum sieťových aplikácií. Základom je možnosť parametricky si zvoliť výsledné vlastnosti, podľa požiadaviek na rýchlosť, vnútorné usporiadanie a kapacitu cache. Flexibilita je tiež podmienená technológiou Virtex-II Pro a implementačným jazykom VHDL, ktoré by mali byť základnou platformou pre návrh a implementáciu.

Práca je členená do niekoľkých kapitol. Na úvod (kapitola 2) je pre prehľadnosť uvedená definícia cache a hlavný princíp, na ktorom je založená jej funkčnosť – hierarchia pamätí a lokalita odkazov. V ďalšej časti sú definície základných pojmov potrebných pre pochopenie následných súvislostí. Na záver kapitoly sú detailne popísané rôzne aspekty návrhu cache pamätí, organizácia dátových štruktúr a algoritmy týkajúce sa rôznych typov cache. Následne v kapitole 3 je navrhnutá požadovaná architektúra, nezávislá na konkrétnej aplikácií a čiastočne aj na použitej implementačnej technológii. Kapitola 4 popisuje výsledky hotovej implementácie. Jedná sa hlavne o zhodnotenie výkonnosti cache (latencia, dĺžka cyklu a priepustnosť), plochy zabratej na čipe a maximálnej frekvencie. Záverečná kapitola diskutuje dosiahnuté výsledky a poukazuje na možnosť návrh do budúcnosti rozšíriť.

Kapitola 2

CACHE pamäte

Cache je malá a rýchla pamäť, používaná obecne vo výpočtových systémoch na ukladanie tých častí hlavnej pamäte, ktoré sa v danom okamihu využívajú, alebo u ktorých je pravdepodobnosť, že sa v blízkej dobe budú využívať na výpočet.

Všeobecne možno povedať, že ideálne parametre pamäte sú maximálna veľkosť a maximálna rýchlosť. Vzhľadom na fakt, že s rastom rýchlosti a kapacity neúmerne rastie aj jej cena, teda vytvorenie veľkokapacitných a rýchlych pamätí je ekonomicky náročné, je potrebné pri voľbe pamäťových zdrojov urobiť kompromis. Jednou z možností je tzv. hierarchia pamätí. Pamäť je rozdelená do niekoľkých vrstiev podľa kapacity a rýchlosti. Na obrázku 2.1 je jednoduchý príklad takejto hierarchie. Najvyššie je umiestnená (centrálna) procesná jednotka – (C)PU[1], spolu s registrami a najrýchlejším prístupom. S každou ďalšou nižšou vrstvou sa znižuje rýchlosť a zvyšuje kapacita. Cieľom je vytvoriť systém, ktorý sa cenou približuje čo najlacnejším druhom pamätí a rýchlosťou čo najrýchlejším.

2.1 Princíp lokality

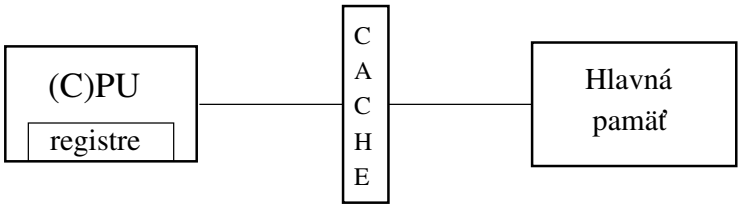
Na uplatnenie hierarchického usporiadania musí platiť tzv. *princíp lokality*. Tento princíp, ktorý je výraznou vlastnosťou programov, je založený na znovupoužití dát. Môžme ho rozdeliť do dvoch typov:

- *Časová lokalita* znamená, že dáta, ktoré boli nedávno použité budú za krátky čas pravdepodobne použité znova.
- *Priestorová lokalita* znamená, že dáta, ktorých adresy sú blízko seba, s pomerne veľkou pravdepodobnosťou budú použité spoločne v krátkom časovom úseku.

Aby teda bolo použitie cache pamäte pre sieťovú aplikáciu efektívne, musí byť k dátam pristupované v blízkych časových úsekoch, poprípade v zhlukoch (priestorová lokalita). V [11] je štúdia lokality IP adries, ktorá potvrdzuje, že *charakteristickým rysom reálnej sieťovej prevádzky je časová lokalita*. Tento fakt je potom využitý na návrh architektúry pre zrýchlenie prístupu do DRAM pamäte pri vyhľadávacích mechanizmoch v IP smerovačoch.

Môžme teda povedať, že použitie cache pamäte má význam aj pre sieťové aplikácie za predpokladu dodržania práve uvedeného princípu.

[1] Práca popisuje cache obecne, preto sa nemusí vôbec jednať o procesor. Na prístupy do pamäte môže byť implementovaná aj jednoduchá procesná jednotka.



Obrázok 2.1: Príklad hierarchie pamätí pre výpočtový systém.

2.2 Základné pojmy

Nasledujúci text na úvod stručne popisuje pojmy definujúce architektúru a vlastnosti cache pamätí. Ku každému názvu je jednoduché vysvetlenie daného aspektu a pojmov s ním súvisiacich.

Cache hit a miss Ak sa pri požiadavke na informáciu zistí, že sa v cache nachádza, nastal tzv. *hit*. Naopak, ak sa informácia v cache nenachádza a je požadovaná, jedná sa o *miss*.

Algoritmus načítavania dát (fetch algorithm) Používa sa na určenie, kedy je potrebné načítať dáta do cache. Existuje niekoľko možností: informácia sa načíta práve vtedy, keď je potrebná alebo dopredu (prefetch). Algoritmy používajúce prefetch sa snažia predpovedať aká informácia bude onedlho potrebná a načítajú ju v predstihu. Ďalšou možnosťou je napríklad vynechanie čítania pri zápise v cache, ktorá používa techniku write-through (vysvetlená neskôr).

Umiestnenie položky (placement algorithm) Cache sama o sebe nie je priamo adresovateľná, ale javí sa aplikácii transparentne. Vzhľadom na to, že je schopná poňať iba zlomok kapacity pamäte na nižšej vrstve, je potrebné zvoliť nejaký algoritmus, ktorý mapuje adresu pamäte na adresu cache. Ďalej častokrát sa informácia vyhľadáva v cache asociatívne. Avšak použitie plne asociatívnej pamäte je jednak drahé a taktiež pomalé, preto sa volí kompromis vo forme tzv. *sady (set)*. Každá sada je tvorená menšou asociatívnou pamäťou. Algoritmus, ktorý určí v ktorej sade sa informácia nachádza a taktiež samotná asociativita sady sú významnými parametrami a v značnej miere ovplyvňujú výkonnosť cache.

Dĺžka riadku Fixne definované množstvo informácií, ktoré sa prenáša medzi cache a pamäťou sa nazýva *riadok*. Zvolenie správnej veľkosti riadku je veľmi dôležitý aspekt návrhu cache pamäte. Iný názov pre riadok v cache je tak isto *blok*.

Algoritmus výberu obete (replacement algorithm) Ak je z pamäte požadovaná informácia a cache je plná, je potrebné zvoliť tzv. *obeť (victim)* a nahradiť ju požadovanou informáciou. Spôsob akým sa obeť určí sa nazýva *nahradzovací algoritmus (replacement algorithm)*. Existuje niekoľko typov, z ktorých najznámejšie sú *LRU (Least Recently Used)*, *FIFO (First In First Out)* a *náhodný výber (Random)*.

Algoritmus zápisu položiek do pamäte (memory update algorithm) Pri zápise dát existuje niekoľko možností ako sa zmena premietne do cache a do samotnej pamäte. Ak sa dáta zapíšu iba do cache a v pamäti ostane stará hodnota, jedná

sa o *write back*. Dáta sa do pamäte zapíšu až vtedy, keď je potrebné ich nahradiť inými. V tomto prípade je zápis pomerne efektívny, ale nevýhodou je, že vzniká nekonzistencia medzi obsahom cache a pamäte. Ďalšou možnosťou je pri zápise dát okamžite zmenu premietnuť aj do pamäte – *write through*. Tento algoritmus odstraňuje problém nekonzistencie, ale platí sa za to určitým zvýšením času cyklu cache, pretože každý zápis vyžaduje prístup do pamäte.

Veľkosť cache Je zrejmé, že veľkosť cache má priamy vplyv na pravdepodobnosť výskytu požadovanej informácie. Avšak veľkosť cache je obmedzená hlavne ekonomickými faktormi. Je to jednak cena, počet zabratých zdrojov a v nemalej miere aj výkonnosť. Preto je dobré už pri návrhu správne analyzovať druh aplikácie a zvoliť primeranú veľkosť cache.

Prístupová doba (latencia) je čas potrebný na vykonanie čítania alebo zápisu, t.j. doba od chvíle keď sa cache poskytne adresa, do chvíle keď sú dáta zapísané alebo pripravené na použitie.

Cyklus pamäte (memory cycle time) je súčet prístupovej doby a času na vykonanie ďalších operácií pred tým ako je možné zahájiť spracovanie nasledujúcej požiadavky. V prípade cache je to čas potrebný na vyčítanie dát z hlavnej pamäte (vtedy sa vstup zablokuje a neprijímajú sa žiadne požiadavky).

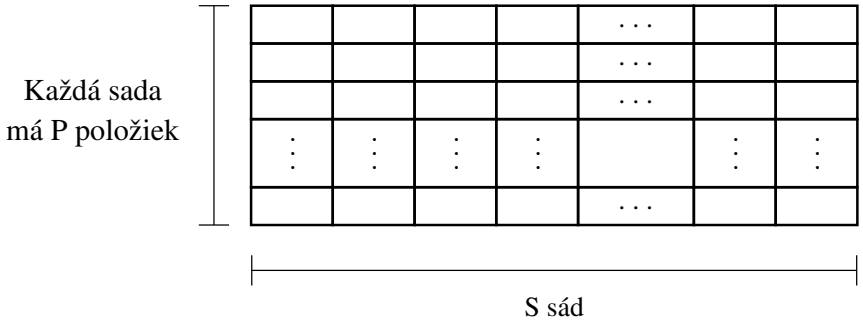
Priepustnosť Jedným z veľmi podstatných faktorov (a hlavne pri sieťových aplikáciách) je priepustnosť, t.j. množstvo informácie za jednotku času, ktoré je možné z cache prečítať. Priepustnosť sa dá zväčšiť hlavne zvýšením *šírky dátovej cesty*, zavedením *zreťazeného spracovania* a taktiež znížením cyklu pamäte (cache).

2.3 Aspekty návrhu cache

2.3.1 Algoritmus umiestnenia položky

Ako už bolo spomenuté, na zistenie, kam uložiť požadovanú položku do cache je potrebné zvoliť nejaký algoritmus. Cache sama o sebe z pohľadu aplikácie nie je priamo adresovateľná a slúži iba ako zásobáreň rýchlo prístupných dát. Algoritmus umiestnenia položky v cache buď zahŕňa nejakú funkciu, ktorá namapuje adresu pamäte do cache, alebo prehľadáva cache asociatívne, poprípade kombinuje tieto dva prístupy. V princípe existujú tri druhy mapovania:

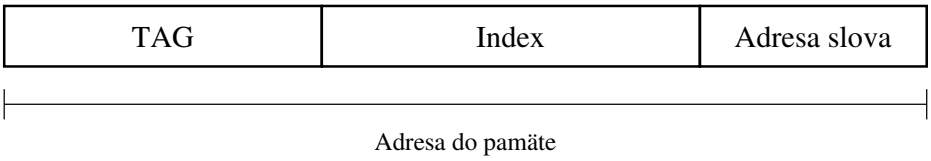
- Ak sa blok (položka) môže nachádzať v cache iba na jedinom mieste, jedná sa o *priamo mapovanú (direct mapped)* cache. V tomto prípade je mapovanie jednoduchý bitový výber.
- Ak sa blok môže nachádzať na ľubovoľnom mieste v cache, hovoríme o *plne asociatívnej (fully associative)* cache.
- Kombináciou týchto dvoch prístupov je *N–cestne asociatívna (N–way set associative)* cache. Cache pozostáva z tzv. sád. *Sada (set)* je skupina blokov v cache. Blok sa najprv namapuje na konkrétnu sadu použitím bitového výberu a potom je možné ho umiestniť kdekoľvek v rámci sady. To znamená že asociatívne sa neprehľadáva celá cache, ale iba daná sada.



Obrázok 2.2: Genericky nastaviteľná organizácia cache pamäte.

Na obrázku 2.2 je znázornené generické usporiadanie cache, ktoré zahŕňa všetky tri vyššie uvedené možnosti. Cache je rozdelená na S sád a každá sada obsahuje P položiek, ktoré sa prehľadávajú asociatívne. Položku je pritom možné stotožniť s riadkom. Celková kapacita C cache je teda daná súčinom: $C = P * S$ blokov. Je zrejmé, že ak zvolíme $S = C$, získame priamo mapovanú cache. Ak zvolíme $P = C$, získame plne asociatívnu cache. V ostatných prípadoch je cache N–cestne asociatívna.

Vzhľadom na to, že na jedno miesto v cache je možné namapovať viacero položiek z pamäte, je potrebné každý blok jednoznačne identifikovať. Na to sa využíva tzv. *tag*. Na obrázku 2.3 je vidieť pozíciu tagu v adrese do pamäte. Časť adresy, ktorá sa použije



Obrázok 2.3: Pozícia tagu v adrese do pamäte. Taktiež je vidieť indexovaciu časť, ktorá sa použije na zvolenie sady pre požadovaný blok.

na indexovanie do cache nie je jednoznačná, preto na jednoznačné určenie adresy bloku sa s dátami musí uložiť aj zvyšná horná časť adresy. Špeciálny prípad nastáva pri plne asociatívnej cache, ktorá má jedinú sadu obsahujúcu všetky riadky. Tam indexovacia časť chýba a preto na jednoznačné určenie riadku v cache je potrebné uložiť celú adresu.

2.3.2 Veľkosť riadku

Veľkosť riadku je jedným z podstatných parametrov, ktoré je potrebné zvoliť pri návrhu. Riadok je základná jednotka pri prenose medzi cache a hlavnou pamäťou. Uplatňuje sa tu blokový prístup, preto sa taktiež nazýva *blok*. Aj keď pojmy riadok a blok sú ekvivalentné, pre jednoduchosť sa v ďalšom texte používa slovo blok.

Možnosť zvoliť pri návrhu veľkosť bloku je veľmi podstatná vec, pretože to ovplyvňuje výkonnosť cache pamäte [9]. Najviac sa veľkosť bloku prejavuje pri miss požiadavkách: ak nastane miss, je potrebné z pamäte načítať celý blok, pričom najhoršia situácia nastáva, keď je cache zaplnená a je potrebné zvoliť obeť. Vtedy je potrebné jeden blok do pamäte zapísať, a jeden z nej prečítať.

Ak je množstvo požadovanej informácie malé, napríklad iba jedno slovo, je výhodné zvoliť malú veľkosť bloku. Čas na nahratie malého bloku z pamäte do cache je určite menší ako keď je blok veľký. Ďalej pri použití krátkeho bloku je menšia pravdepodobnosť, že bude obsahovať informácie, ktoré sa nepoužijú.

Na druhej strane, ak sa využíva väčšia časť bloku, ideálne celý, jeho zväčšením sa môže dosiahnuť zvýšenie výkonu. Pri načítavaní z pamäte sa informácia načíta naraz, čo je oveľa efektívnejšie. Väčšie bloky taktiež znižujú počet adresovaných položiek v cache a ten znižuje počet logických obvodov potrebných na implementáciu.

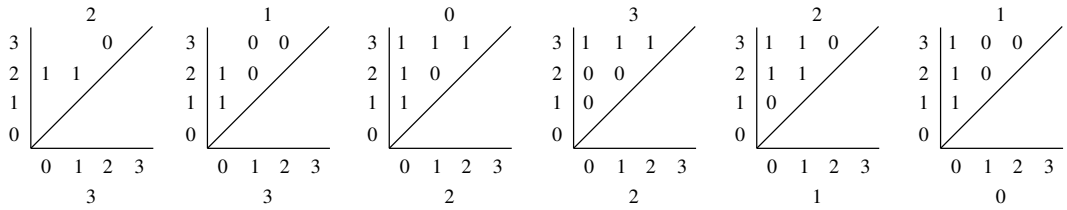
Je teda zjavné, že výhody jedného prístupu sú automaticky nevýhody toho druhého, preto možnosť zvoliť si veľkosť bloku podľa požiadaviek konkrétnej aplikácie je veľmi podstatná vec a mala by byť jedným z nastaviteľných parametrov generickej cache.

2.3.3 Spôsob výberu obete

Keď nastane miss a cache je plná, je potrebné zvoliť blok, ktorý sa nahradí požadovanými dátami. Algoritmus výberu obete definuje spôsob, akým sa obeť vyberá.

Ak sa na výber použije náhodný generátor, jedná sa o *náhodný výber (random)*. V praxi sa pre jednoduchosť používa pseudonáhodné generovanie kandidáta. V prípade, že sa na výber použije spôsob fronty, hovoríme o *FIFO* algoritme. Obidve možnosti sa implementujú pomerne jednoducho buď použitím čítača, alebo nejakej formy generátora pseudonáhodných čísel, s periódou N, kde N je počet položiek na výber obete. Ďalšou možnosťou je použiť princíp tzv. najneskôr použitej položky – *LRU (Least Recently Used)*. Tento algoritmus je veľmi populárny pre jeho vyššiu výkonnosť oproti FIFO a náhodnému výberu [9]. Jeho nevýhodou je problematická hardwarová implementácia. Existuje niekoľko spôsobov ako tento algoritmus implementovať, každý z nich sa líši precíznosťou a nárokmi na zdroje.

Implementácia plného LRU je pamäťovo náročná – algoritmus navrhnutý v [9] využíva tzv. *referenčnú maticu* a pre sadu obsahujúcu N položiek spotrebuje $N(N-1)/2$ stavových bitov. Na uloženie bitov sa vytvorí ľavá horná trojuholníková matica bez diagonály ($i+j <$

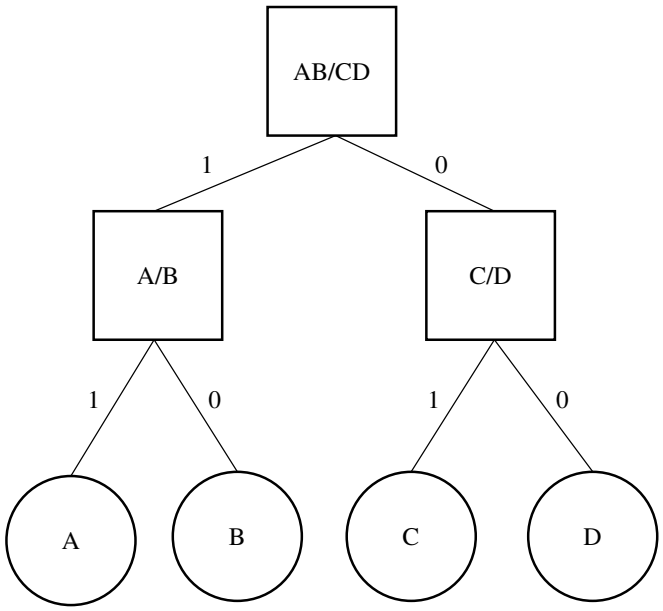


Obrázok 2.4: Príklad LRU použitím referenčnej matice. Vrchné hodnoty znamenajú index aktualizovanej položky. Spodné hodnoty určujú LRU obeť.

N, kde i a j je index riadku, resp. stĺpca) s rozmermi $N * N$. Pri prístupe na položku i sa všetky bity v riadku i nastavia na jedna a všetky bity v stĺpci i matice sa nastavia na nula. LRU obeť je tá, ktorá vo svojom riadku má všetky bity nulové a v stĺpci má všetky bity jednotkové (obrázok 2.4). Tento postup je pre 4 položky pomerne výhodný, je treba vyhradiť 6 bitov. Pre 8 položiek je už treba 28 bitov, čo je ešte realizovateľné, ale pre 16 a viac položiek je už tento postup nevhodný, pretože vyžaduje aspoň 120 bitov.

Sú známe aj iné metódy implementácie plného LRU algoritmu, napríklad v [13] je implementovaný obojsmerne viazaný zoznam. V [4] je navrhnuté použiť systolické pole pre

ľubovoľné množstvo položiek, za cenu pomerne veľkej pamäťovej náročnosti. Je zjavné, že pamäťová náročnosť pri použití zoznamu je lineárna, avšak platí sa za to nemožnosťou vykonať zmenu položky v jednom hodinovom cykle (pre cache je to veľmi podstatná vec, pretože každým cyklom naviac sa zvyšuje doba odozvy). Tento problém naopak rieši systolické pole, avšak s veľkými pamäťovými nárokmi.



Obrázok 2.5: Príklad pseudo LRU algoritmu pre štyri položky použitím binárneho stromu.

Existujú preto algoritmy, ktoré LRU iba aproximujú, sú však jednoduchšie. Jedným z nich je *pseudo LRU* (a jeho modifikácie [6]), ktorý na uchovanie stavovej informácie používa binárny strom. Príklad takéhoto stromu pre 4–cestne asociatívnu cache je na obrázku 2.5. Jednotlivé bloky sú reprezentované listmi stromu – A, B, C a D. História prístupov je daná tromi bitmi v uzloch - A/B, C/D a AB/CD. Pri prístupe na jeden z dvojice blokov A a B sa nastaví bit AB/CD na 1, pri prístupe na jeden z blokov C a D sa AB/CD bit nastaví na 0. Podobne je to pri bitoch A/B a C/D. Postup zmeny konkrétnych bitov je v tabuľke 2.1. Pseudo LRU je iba aproximácia pravého LRU, preto aj zvolené LRU položky sú iné. Tabuľka 2.2 porovnáva rozdiel medzi LRU a pseudo LRU počas série niekoľkých referencií.

Blok	AB/CD bit	A/B bit	C/D bit
A	1	1	bez zmeny
B	1	0	bez zmeny
C	0	bez zmeny	1
D	0	bez zmeny	0

Tabuľka 2.1: Zmena bitov pri prístupe na jednu z položiek v pseudo LRU algoritme.

2.3.4 Zápis do pamäte

Ak je potrebné do cache zapísať dáta, existujú dve možnosti ako sa zachovať:

Čas	1	2	3	4	5	6	7	8	9	10
Prístup na položku	A	D	B	C	B	A	D	A	B	C
PLRU	D	B	C	A	D	D	B	C	C	A
LRU	D	C	C	A	A	D	C	C	C	D

Tabuľka 2.2: Rozdiely zvolenej LRU položky pri LRU a pseudo LRU.

- *Write through* – Jedná sa o jednoduchšiu techniku, kde všetky požiadavky na zápis ovplyvňujú cache a aj samotnú pamäť. To zabezpečuje, že dáta v hlavnej pamäti sú vždy platné. Tento prístup sa najviac uplatní pri multiprocesorových systémoch, kde je potrebné jednotlivé dvojice procesor-cache synchronizovať so zmenami v hlavnej pamäti. Jeho nevýhodou je, že každý zápis do cache spôsobí aj zápis do pamäte, preto môže značne ovplyvniť jej výkonnosť.

- *Write back* – Táto technika minimalizuje zápisy do pamäte, poskytujúc takto potenciálne vyšší výkon. Jedná sa ale na druhej strane o zložitejšiu a na zdroje náročnejšiu variantu. Ak je cache plná a je potrebné vymeniť niektorý blok, musí byť známe či je pamäťový blok s ním korešpondujúci platný. Ku každému bloku v cache je teda potrebné vyhradiť jeden bit naviac aby udával jeho zneplatnenie (invalidate). Tento bit sa všeobecne nazýva *dirty bit*. Ak je potrebné z cache odstrániť položku musí sa tento bit rešpektovať a pri jeho nastavení daný blok zapísať do pamäte.

V prípade, že pri zápise nastane miss, nemusí byť vždy nutné v cache alokovať priestor pre požadovaný blok. Sú dve možnosti:

- *Write allocate* – Pri zápisovej miss požiadavke sa pre požadovaný blok vyhradí priestor a blok sa nahrá z pamäte do cache. Teda miss požiadavky pri zápise sa chovajú rovnako ako pri čítaní. Systém write allocate je výhodný ak je predpoklad, že po zápise bude krátko na to nasledovať čítanie, preto sa oplatí daný blok v cache alokovať.

- *No-write allocate* – Jedná sa o opačný prípad, kde zápisové operácie neovplyvňujú cache. Namiesto toho blok je modifikovaný priamo v pamäti.

Každá z vyššie uvedených variant má svoje výhody a nevýhody a je dobré ju použiť v rôznych prípadoch.

2.3.5 Veľkosť cache

Veľkosť cache pamäte je ďalším z parametrov, ktoré vo významnej miere ovplyvňujú jej výkonnosť. Aj tu platí v zásade, že s rastúcou veľkosťou klesá podiel miss požiadaviek a rastie jej cena, počet logiky na adresovanie, atď. Preto platí, že väčšie cache sú v porovnaní s malými o niečo pomalšie (dokonca, ako ukazuje sekcia 4.2.2, veľkosť ako jeden z parametrov môže prinášať značné rozdiely frekvencie, spôsobenej práve množstvom kombinačnej logiky potrebnej na adresovanie). Príliš veľká cache by preto nemala byť na úkor prístupovej doby, ale zase príliš malá cache výrazne degraduje výkonnosť celého systému. Je teda potrebný kompromis medzi požiadavkami a cenou.

Kapitola 3

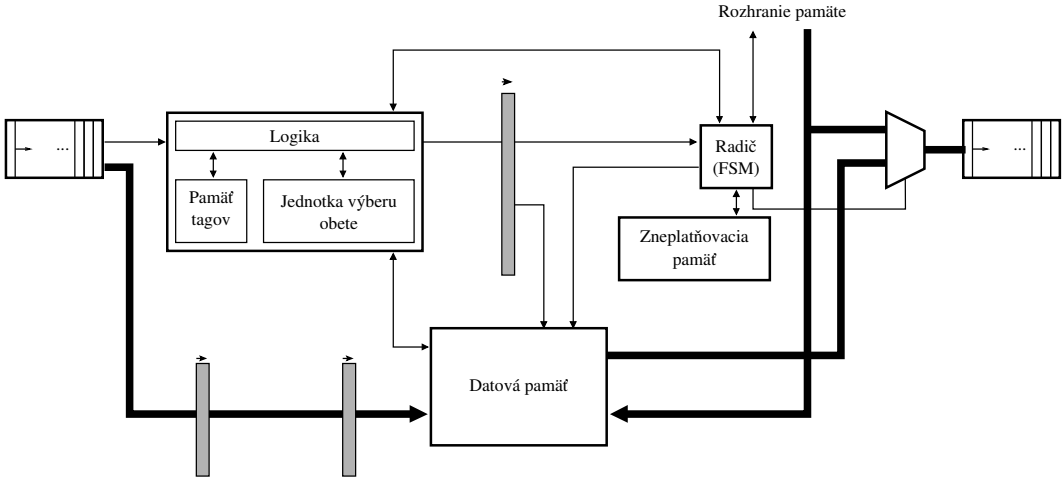
Architektúra

Hlavnou požiadavkou na funkčnosť cache je jej použiteľnosť pre široké spektrum sieťových aplikácií, ale aj iných aplikácií, kde je potrebné rýchlo pristupovať k pomalej pamäti. Ako bolo popísané v kapitole 2, existuje množstvo parametrov, ktoré vplývajú na jej charakteristiku. Pri návrhu bol kladený dôraz hlavne na nasledujúce parametre:

- **Dátová šírka slova (DS)** udáva najmenšiu jednotku adresovania medzi cache a hlavnou pamäťou. Dátová šírka je taktiež mienená ako najmenšia adresovateľná časť medzi aplikačnou jednotkou a cache. Tým sa docieli transparentnosť.
- **Veľkosť bloku (VB)**, ktorú je možné stotožniť s veľkosťou riadku, udáva množstvo informácie, ktoré si predávajú navzájom cache a hlavná pamäť pri čítaní, resp. zápise. Veľkosť bloku viac ako jedno slovo sa uplatní hlavne ak aplikačná jednotka potrebuje pracovať s väčším objemom dát, ale nedokáže ho adresovať v jednom kroku.
- **Asociativita (A)** udáva počet blokov v jednej sade (sekcia 2.3.1). Cache by mala byť schopná realizovať asociativitu jedna až N, kde maximálne N bude obmedzené implementačnými možnosťami. Prípad jediného plne asociatívneho riadku sa neuvažuje. Vzhľadom na náročnosť implementácie CAM pamäte prináša plne asociatívna cache veľmi malý prínos.
- **Veľkosť cache (VC)** udáva počet blokov, ktoré obsahuje jedna cesta. Táto veľkosť je udaná iba v jednom rozmere. To znamená, že celkový počet blokov v cache V_B bude daný vzorcom
$$V_B = (\text{Veľkosť cache}) \times \text{Asociativita}$$
- **Algoritmus výberu obete** udáva spôsob akým sa zvolí obeť v prípade zaplnenia cache. Je potrebné implementovať tri najznámejšie spôsoby: *LRU*, *FIFO* a *náhodný výber*.
- Spôsob zápisu typu **write back** a **write allocate** pre minimálnu prevádzku medzi cache a hlavnou pamäťou.
- Podpora **ľubovoľnej adresovej šírky ($\mathbf{A_P}$)** hlavnej pamäte. Dĺžka adresy bude zadávaná ako jeden z parametrov.
- Podpora **zreťazeného spracovania**.

Na obrázku 3.1 je znázornená bloková štruktúra. Vstupná fronta vyrovnáva výkonnostné rozdiely medzi aplikačnou jednotkou a cache. Ukladajú sa do nej požiadavky na čítanie alebo zápis, spolu s potrebnými dátami (pri zápise). Požiadavka sa následne presúva do *bloku tagov*, kde sa vyhodnotí, či sa požadovaná položka v cache nachádza a zároveň sa daná položka namapuje – určí sa adresa a poloha v sade v dátovej pamäti. Blok tagov zahrňuje *pamäť tagov*, ktorá slúži na zistenie, či sa požadovaná informácia v cache nachádza alebo nie a *blok výberu obete*. Ten pri každej požiadavke aktualizuje stav histórie prístupov a neustále poskytuje na výber obeť, ktorá sa použije v prípade zaplnenia cache.

V ďalšej fáze sa podľa adresy vyčítajú samotné dáta a nakoniec sa podľa získaných informácií spracuje hit/miss požiadavka. Na riadenie celého systému slúži *radič* vo forme stavového automatu. Radič úzko spolupracuje so *zneplatňovacou pamäťou (dirty bit memory)*. Tá slúži na uchovanie informácie o zápise do bloku. Pre každý blok v cache je teda potrebný jeden bit, ktorý sa pri zápise do bloku nastaví. Ak je pri výbere obete tento bit nastavený, je nutné obeť zapísať do hlavnej pamäte, pretože položka v nej je zneplatnená. Tento postup platí iba pri zápisoch typu write-back, kde môže nastať nekonzistencia medzi dátami v cache a v hlavnej pamäti.



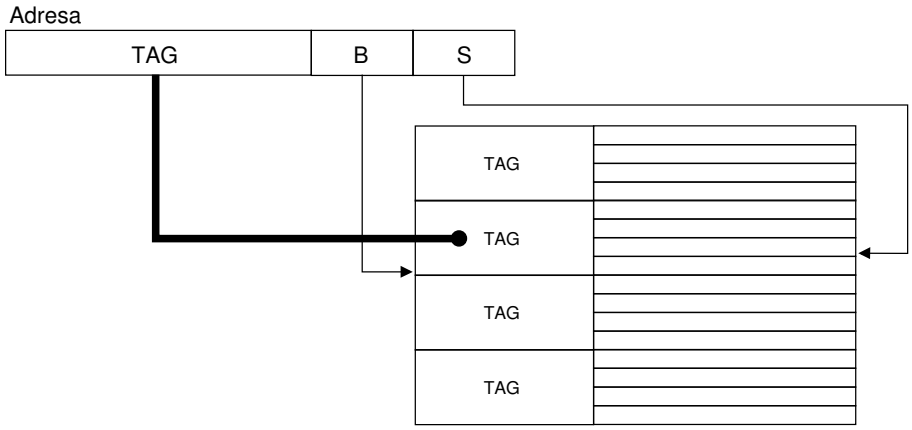
Obrázok 3.1: Bloková štruktúra cache

Vzhľadom na to, že cache podporuje zreťazené spracovanie, informácia o tom, či sa požadovaná položka nachádza v cache (nastal hit), sa propaguje cez jednotlivé stupne reťazca. Zisťovanie, či nastal hit alebo miss sa získava z pamäti tagov. Tak isto pri vytvorení alebo nahradení položky v pamäti tagov (teda nastal miss) je potrebné pristupovať do tejto pamäte. Z obrázka 3.1 vyplýva, že v prípade ak sa v reťazci spracovania nachádza informácia o položke, ktorá sa práve prepísala v pamäti tagov, táto informácia sa zneplatní vo všetkých stupňoch reťazca. Je preto výhodné implementovať minimálne množstvo stupňov zreťazeného spracovania, u ktorých je nutnosť zneplatnenia. V súčasnom návrhu je iba jeden takýto stupeň, implementovaný ako register, pričom do budúcnosti sa počíta s tým, že bude voliteľný. Register vložený do tohoto spracovania zvyšuje výslednú frekvenciu cache, takže pri vyšších stupňoch asociácie je výhodou. Naopak, pri priamo mapovanej cache, kde je možné dosiahnuť pomerne vysoké frekvencie, je tento register zbytočný, pretože zvyšuje výslednú latenciu.

Jednotlivé bloky sú implementované genericky. Ich vnútorné usporiadanie a vlastnosti je možné meniť pomocou parametrov, ktoré sú súčasťou rozhrania.

3.1 Adresovanie

V cache je potrebné adresovať jednotlivé riadky, a keď je asociativita viac ako jedna, adresou je potrebné určiť aj pozíciu bloku v rámci riadku. Väčšina pamätí má svoj obsah zakódovaný binárne a využíva svoju rýchlu vnútornú logiku na určenie aktuálnej bunky – na adresáciu riadku je v tomto návrhu použité binárne kódovanie, pretože je predpoklad, že každá technológia obsahuje nejaké rýchle pamäťové bloky, kde vstupom je iba adresa. Na mapovanie riadku v cache sa využíva bitový výber (obrázok 3.2). Adresa do pamäte



Obrázok 3.2: Adresovanie riadku v cache. S - adresa slova, B - adresa bloku, TAG - jednoznačný identifikátor bloku.

sa rozdelí na niekoľko častí. Ich šírka závisí od generických parametrov: $A_S = \log_2(\text{VB})$ udáva počet bitov na adresovanie slova v rámci bloku, $A_B = \log_2(\text{VC})$ udáva adresu bloku v cache. Na presnú identifikáciu sa zvyšok adresy uloží ako tag do pamäte tagov. Šírka tagu je teda závislá na veľkosti hlavnej pamäte, cache a bloku v cache, podľa nasledujúceho vzťahu:

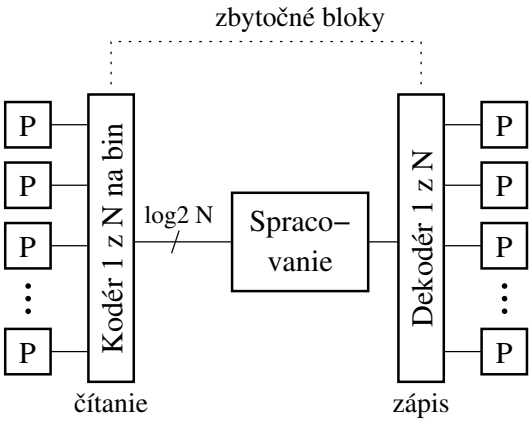
$$A_{\text{TAG}} = A_P - A_B - A_S$$

Na určenie adresy bloku v rámci riadku (dá sa považovať za druhý rozmer, pozri sekciu 3.2) sa použije kódovanie "1 z N". Na rozdiel od binárneho prináša v niektorých prípadoch väčší výkon za cenu širších adresových zberníc. V tomto návrhu je výhodný hlavne z hľadiska asociatívneho prehľadávania pamäti tagov - kde vyhľadaním sa získa *práve jedna položka* z celkového počtu blokov v danej sade. Na binárne zakódovanie po asociatívnom vyhľadaní položky by bol potrebný kodér a naopak, pri zápise, kde je potrebné vybrať podľa adresy práve jeden z blokov v sade, je potrebné znova použiť dekodér. Ako je znázornené na obrázku 3.3 a 3.4, pri adresovaní "1 z N"odpadá nutnosť použiť kodér a dekodér, čím sa môže výrazne zlepšiť výsledné časovanie.

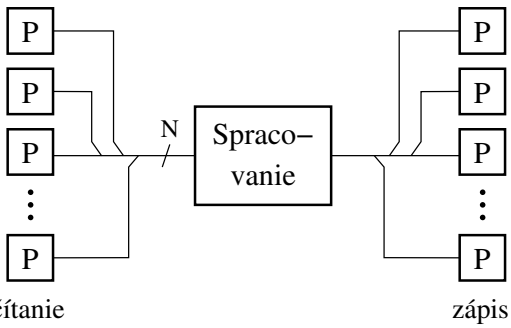
3.2 Dvojrozmerné pamäte

Ak je asociativita cache viac ako jedna, je možné dáta vnímať ako maticu s rozmermi $V * A$, kde V značí počet riadkov a A značí asociativitu. Jednou zo základných stavebných blokov návrhu je teda dvojrozmerná pamäť. Je potrebná v týchto prípadoch:

- Uskladnenie dát a TAG jednotiek.



Obrázok 3.3: Asociatívne určená adresa zakódovaná binárne.



Obrázok 3.4: Asociatívne určená adresa zakódovaná ako "1 z N". Nie je nutný kodér a dekodér.

- Uskladnenie informácií pre výber obete, histórie zápisu do bloku (riadku), zneplatňovacia pamäť.

Je zrejmé, že tieto časti výraznou mierou vplývajú na celkovú latenciu výslednej cache pamäte. Samotný návrh sa nijako neviaže na implementačnú technológiu, požaduje však maximálne latencie pamäťových blokov. V prvom prípade, kedže sa jedná o uskladnenie veľkého objemu informácie, je povolená latencia maximálne jeden hodinový cyklus. V druhom prípade je potrebné vytvoriť pamäte, ktoré pracujú bez hodinovej synchronizácie – ako kombinačná logika. Táto nutnosť plynie z požiadavky zreťazeného spracovania, cieľom je schopnosť cache spracovať každý hodinový cyklus čítanie/zápis jedného slova. To znamená, že každý hodinový cyklus je potrebné zapísať do pamäte položku a zároveň položku prečítať. Použitie synchronizovanej pamäte by si v tomto prípade vyžiadalo buď zvýšenie latencie, alebo nutnosť zložito takúto pamäť implementovať.

Generická schéma dvojrozmernej pamäte je na obrázku 3.5. Na adresáciu slúžia dva vstupy: *adresa slova* a *adresa cesty*. Adresa slova je kódovaná binárne a adresa cesty je kódovaná ako "1 z N". Výstup z pamäte je realizovaný dvoma spôsobmi:

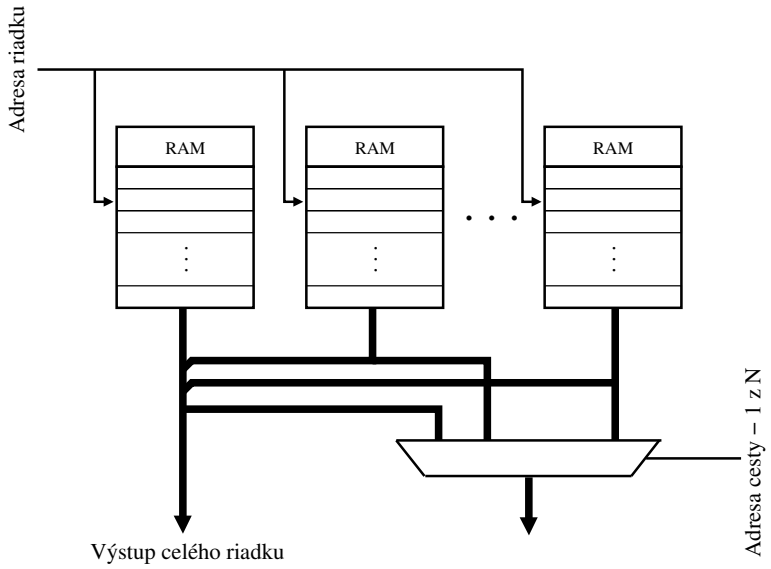
- Výstup jedinej zvolenej položky – pre určenie položky je potrebné vedieť obe adresy. Táto možnosť sa použije pri priamom adresovaní.
- Výstup celého riadku – použije sa, ak je potrebné získať všetky dáta z riadku, napríklad pri asociatívnom prehľadaní (sekcia 3.3).

Zápis prebieha podobným spôsobom: Adresa riadku je zakódovaná binárne, pričom na adresáciu stĺpca nie je potrebný dekodér.

Na implementáciu týchto dvojrozmerných pamätí boli použité dva typy pamäťových buniek technológie Virtex-II Pro (kompletnú špecifikáciu je možné nájsť v [12]).

Pamäť pre výber obete, histórie zápisu do bloku a zneplatňovacia pamäť vyžadujú minimálnu latenciu, preto bola použitá Distribuovaná SelectRAM+ (DistRAM). Každá DistRAM jednotka implementuje 16x1 bitovú pamäť so synchronizovaným zápisom a kombinačným čítaním. Na vytvorenie väčších kapacít je samozrejme možné DistRAM kaskádne spojovať a vytvárať tak, ako hovorí názov, pamäť distribuovanú na ploche FPGA.

Na implementáciu pamätí pre uskladnenie samotných dát a tagov už DistRAM nie je vhodná. Nie je totiž možnosť efektívne pospájať distribuované bloky. Preto sa použila tzv. *Bloková SelectRAM+ pamäť (BRAM)*. Ako napovedá názov, pamäť je organizovaná



Obrázok 3.5: Schéma dvojrozmernej pamäte.

do blokov, ktoré poskytujú kapacitu až 18Kb. BRAM je takisto možné kaskádne spojovať a vytvárať tak pomerne veľké pamäte.

Použitá technológia umožňuje takto vytvoriť dvoj-portové pamäte. Blokové pamäte obsahujú dva úplne na sebe nezávislé porty určené na čítanie alebo zápis (to využíva blok tagov, sekcia 3.5). Distribuované pamäte poskytujú jeden port na čítanie alebo zápis a jeden port iba na čítanie. V prípade potreby je možné jeden z portov odpojiť a minimalizovať tak množstvo zabratých zdrojov.

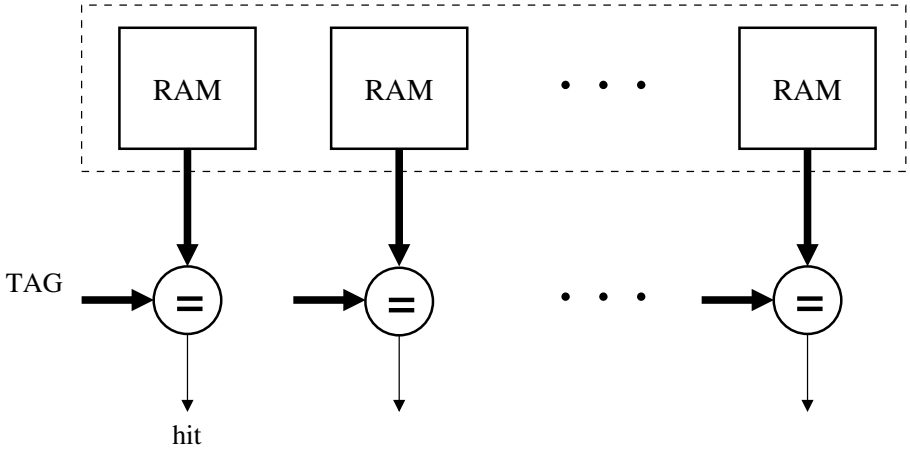
3.3 Pamäť tagov

Pamäť tagov zabezpečuje dve základné funkcie:

- Asociatívne vyhľadanie bloku v sade. Konkrétna sada sa namapuje podľa spôsobu adresovania (sekcia 3.1).
- Vyhľadanie voľnej položky v sade. V prípade, že požadované dáta sa v cache nenachádzajú, je potrebné zistiť, či je v danej sade voľná položka, a ktorá to je.

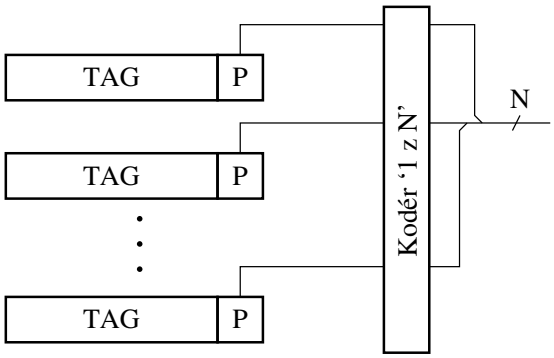
Ako už bolo spomenuté, na jednoznačné určenie, či sa blok nachádza v cache sa musí použiť tag odvodený z adresy do hlavnej pamäte. Tag sa ukladá spolu s bitom, ktorý udáva jeho platnosť, do dvojrozmernej pamäte. Na asociatívne prehľadanie pamäte sa využije séria komparátorov (obrázok 3.6). Aby tento systém pracoval správne, práve jeden zo všetkých komparátorov musí byť aktívny v prípade, že požadovaný blok sa nachádza v sade. Na zachovanie konzistencie preto musí každá sada vždy obsahovať rozdielne tagy. V opačnom prípade sa poruší princíp kódovania “1 z N”a cache sa dostane do nedefinovaného stavu. Túto konzistenciu zaručuje radič, ktorý vyhodnocuje výsledky vyhľadania.

V prípade, že je potrebné vytvoriť v cache novú položku, musí sa vybrať v rámci sady tá, ktorá ja voľná. Ak predpokladáme, že bit platnosti v pamäti tagov udáva voľnú položku, je možné z neho odvodiť jej adresu - ak je bit nastavený, udáva platnú položku, v opačnom



Obrázok 3.6: Komparátory implementujúce asociatívnu pamäť pre vyhľadanie bloku v sade.

prípade je voľná a môže sa použiť. Na rozdiel od identifikátorov bloku v rámci sady (tag), aj niekoľko takýchto bitov môže byť v tejto sade nastavených na nula (položka je voľná). Na zakódovanie je preto použitý prioritný kodér, ktorý zakóduje danú postupnosť ako "1 z N"(obrázok 3.7).



Obrázok 3.7: Určenie voľného bloku v cache pomocou generického prioritného kodéru "1 z N".

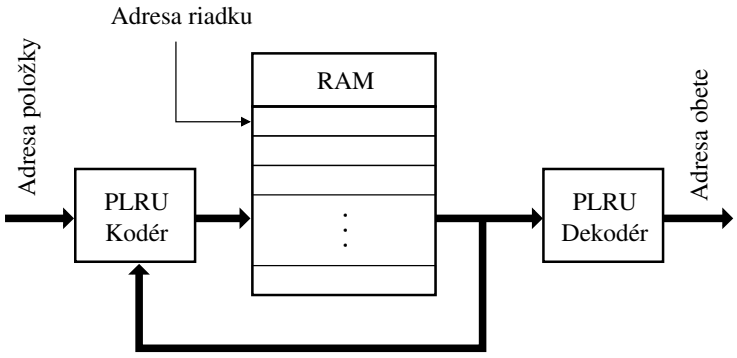
3.4 Jednotka výberu obete

Jednotka výberu obete sa použije iba v prípade, že asociativita je viac ako jedna. Ak sa cache zaplní a v sade je potrebné založiť novú položku, musí sa zvoliť obeť a tá z cache odstrániť. Na tento blok sú kladené prísne požiadavky časovania, pretože v každom hodinovom cykle musí byť schopný aktualizovať informáciu pre výber obete. Je preto vhodné, aby logika, ktorá to realizuje, bola čo najjednoduchšia a zaberala čo najmenej miesta.

Na implementáciu LRU bol vybraný algoritmus *pseudo LRU (PLRU)*, ktorý síce nie je úplne presný, ale má veľmi malú pamäťovú zložitosť. Na určenie PLRU z N položiek je potrebných $N-1$ bitov. Pri tomto algoritme na uskladnenie informácií o prístupe pre celú cache je treba VC $*$ $(N-1)$ bitov. Pre porovnanie – pri použití referenčnej matice

by pre celú cache bolo nutné vyhradiť VC $* N(N-1)/2$ bitov. Pre väčšie asociativity sú pamäťové nároky neprípustné, vzhľadom na kapacity dnešných technológií [12].Ďalšou z výhod PLRU algoritmu je jeho stromová štruktúra – na zakódovanie histórie prístupov na položky v rámci sady sa využíva binárny strom. Pre N položiek je výška stromu $\log_2 N$, čo je aj zároveň cesta potrebná na vyhľadanie LRU položky. Pri hardwarovej implementácii to znamená, že na vyhľadanie LRU položky je potrebná funkcia $\log_2 N$ vstupov. Pre tieto priaznivé vlastnosti – lineárna pamäťová a logaritmická časová zložitosť, je PLRU výhodný algoritmus, poskytujúci porovnateľné výsledky s LRU.

Obrázok 3.8 znázorňuje použite pamäte, kódovacieho a dekódovacieho obvodu PLRU. V RAM pamäti sú uložené konfiguračné bity uzlov binárneho stromu. Kódovací obvod slúži na zmenu konfigurácie stromu podľa toho, na akú položku bolo práve pristupované. Dekódovací obvod realizuje určenie LRU položky – prehľadanie binárneho stromu. Pre akékoľvek hodnoty stavových bitov v strome bude LRU aktívna práve jedna položka. Keďže adresa bloku v sade je zakódovaná ako "1 z N", je určovanie adresy LRU bloku jednoduchá (a hlavne rýchla) AND funkcia s $\log_2 N$ vstupmi pre každý z N možných blokov. Pre každý blok totiž daná AND funkcia kopíruje svoju jednoznačnú cestu v binárnom strome. Na



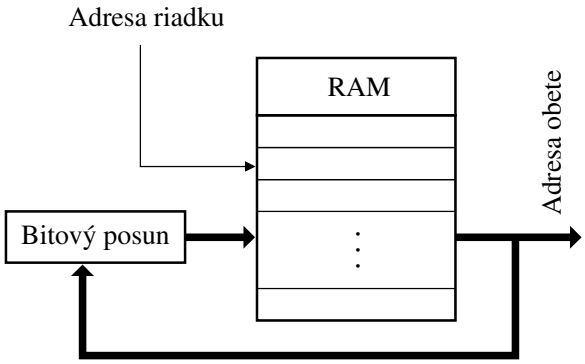
Obrázok 3.8: Implementácia PLRU s použitím pamäte na uskladnenie histórie.

obrázku je možné si všimnúť spätnú väzbu, smerujúcu z výstupu pamäte na vstup PLRU kodéru. Pri aktualizácií histórie prístupov v RAM je totiž nutné zachovať pôvodné hodnoty niektorých bitov v uzloch stromu. V tomto prípade existujú dve možnosti ako zapísať do pamäte bity jednotlivo:

- Pamäť podporuje maskovanie zápisu. Spätná väzba nie je potrebná, bity, ktoré pri zápise ostávajú bez zmeny sa určia maskou.
- Pamäť nepodporuje maskovanie zápisu. Je nutné najprv vyčítať celé slovo, z neho vybrať nezmenené bity a následne pozmenené slovo zapísať.

Nutnosť spracovať každý hodinový cyklus jednu požiadavku a potreba spätnej väzby vyžadujú aby sa RAM použitá na PLRU algoritmus chovala ako kombinačná logika.

Na implementáciu FIFO výberu obete je možné použiť jednoduchý čítač s periódou N, kde N je počet položiek v sade. Keďže adresa je zakódovaná ako "1 z N", je výhodné namiesto binárneho čítača použiť rotujúcu jedničku. To sa docieli bitovým posunom výstupnej hodnoty RAM a následným zápisom na rovnakú adresu (obrázok 3.9). Bitový posun sa aktivuje pri každom výbere obete. Tým sa docieli, že ďalšia obeť bude vždy na nasledujúcej adrese.



Obrázok 3.9: FIFO výber obete s použitím rotujúcej jednotky.

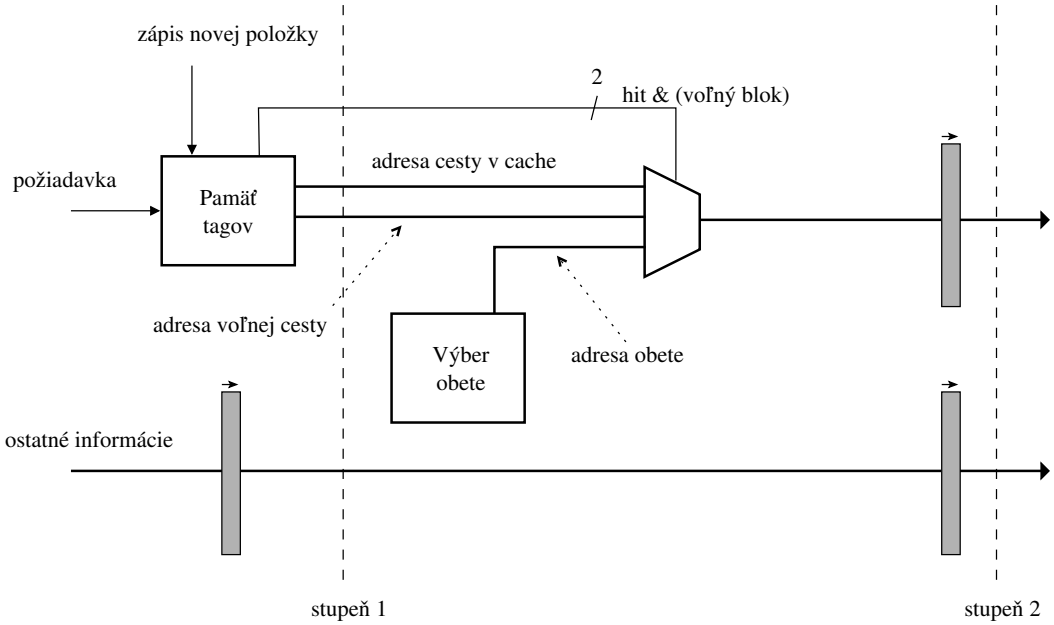
Asi najjednoduchším a najlacnejším spôsobom je náhodný výber obete. V tomto prípade nie je nutné použiť RAM. Postačujúci je jediný register implementujúci rotujúcu jednotku každý hodinový cyklus [9]. Keďže prístupy do cache nie je možné predvídať, dá sa tento spôsob považovať sa určitú jednoduchú formu náhodného generátoru.

3.5 Blok tagov

Táto jednotka zapuzdruje pamäť tagov a blok výberu obete. Má dve rozhrania. Rozhranie na čítanie je napojené na vstup cache (vstupná fronta) a smeruje do pamäti tagov, kde sa vyhodnotí, či nastal hit alebo miss a zistí sa či je v cache voľná položka a ktorá to je. Táto informácia sa použije na výber adresy v rámci bloku, podľa postupu popísanom v sekcii 3.6. Rozhranie na zápis slúži na vkladanie informácií o nových položkách v cache. Obidve rozhrania sú adresované osobitne, a ak je možné použiť dvojportové pamäte, tak nie je nutné multiplexovať adresy.

Obrázok 3.10 popisuje usporiadanie jednotlivých podkomponent v rámci jednotky. Prerušované čiary zobrazujú stupne zreťazeného spracovania. Ako prvý stupeň slúži pamäť tagov, v druhom stupni sa spracuje jej výsledok a získané informácie sa posielajú do ďalších častí cache. Blok tagov poskytuje pre radič priebežné výsledky z oboch stupňov. Takto sa zabezpečí plynulosť spracovania požiadaviek – a tým je splnená podmienka, že cache musí zvládať spracovať každý hodinový cyklus jedno slovo (samozrejme v prípade, že sa jedná o hit).

Hlavnou výhodou zreťazeného spracovania je možnosť použitia vyšších frekvencií, hlavne v prípade veľkej asociativity. Ako je vidieť na obrázku 3.10, blok tagov na výstupe obsahuje register, ktorý zabezpečuje práve funkciu zreťazeného spracovania. Ak však na konkrétnu aplikáciu postačuje malá a jednoduchá cache (napríklad priamo mapovaná cache pre 128 položiek, s veľkosťou bloku jedna), bolo by výhodné mať možnosť tento register vypustiť, a tak znížiť celkovú latenciu. Bohužiaľ súčasný návrh toto neumožňuje, do budúcnosti sa však počíta s možnosťou voliteľnej latencie a tým väčšou kontrolou uživateľa nad architektúrou cache.



Obrázok 3.10: Schéma bloku tagov.

3.6 Radič cache

Na generovanie riadiacich signálov pre jednotky slúži stavový automat. V princípe je možné na riadenie každej časti cache vytvoriť samostatný automat, pričom by jednotlivé automaty navzájom komunikovali. Nejedná sa však o až tak zložitý systém, ktorý by toto striktne vyžadoval v záujme zachovania prehľadnosti návrhu. Preto bol pre celú cache navrhnutý jediný, aj keď o niečo zložitejší, radič, ktorý spracováva informácie o všetkých jednotkách a na základe toho generuje riadiace signály. Automat v priebehu svojej činnosti prechádza niekoľkými fázami:

1. Inicializácia. Na začiatku je potrebné vynulovať pamäte, ktoré reprezentujú platnosť – pamäť tagov a zneplatňovacia pamäť. Počas behu inicializácie cache ešte nie je schopná spracovávať požiadavky.

2. Získanie výsledku z bloku tagov – z jednotlivých stupňov bloku (sekcia 3.5) sa vyhodnotí, či sa jedná o zápis alebo čítanie. Podľa toho sa činnosť stavového automatu delí na čítaciu a zápisovú vetvu. Pre každú vetvu existuje podvetva pre spracovanie hit alebo miss požiadavky. Informácia, či sa jedná o hit alebo miss sa tak isto získava z bloku tagov.

3. V prípade, že sa jedná o hit (požadovaná položka sa nachádza v cache), jednoducho sa urobí zápis/čítanie, podľa toho v akom stave sa automat práve nachádza. Následne automat prejde naspäť na krok 2 a začne vykonávať ďalšiu predspracovanú požiadavku.

4. Ak sa jedná o miss (položka v cache nie je), vytvorí sa požiadavka na prečítanie požadovaného bloku z hlavnej pamäte, založí sa nová položka. Po zaplnení cache je potrebné zvoliť obeť. Radič však už samotné zvolenie obete nevykonáva – o to sa

v predchádzajúcich fázach postarala jednotka výberu obete, umiestnená v bloku tagov. Zápis obete (bloku) do pamäte sa udeje iba keď je v hlavnej pamäti zneplatnený. Informácia o konzistencii bloku vybratého na zápis sa získa zo zneplatňovacej pamäte.

5. Cache následne čaká na prijatie bloku z hlavnej pamäte, ktorý sa zapíše do dátovej časti.

6. Dokončí sa požiadavka – podľa toho v akej vetve sa automat nachádza, sa vykoná buď zápis alebo čítanie.

7. Radič pokračuje v práci od bodu 2.

Radič pracuje ako Mealyho stavový automat. V hardwarovej implementácii každý prechod medzi stavmi obecne znamená jeden hodinový cyklus. Mealyho typ, keďže minimalizuje expanziu stavov, umožňuje vytvoriť čo najviac úsporný model. Tým sa skracuje čas potrebný na spracovanie požiadavky. Na druhej strane Mealyho výstup, ako funkcia aktuálneho stavu a aj vstupov, kladie zvýšené nároky na kombinačnú logiku výsledného automatu. To v niektorých prípadoch môže znížiť maximálnu frekvenciu cache.

Kapitola 4

Výsledky implementácie

S ohľadom na požiadavky na generickosť, použitú technológiu a jazyk bola vytvorená konečná implementácia. Táto kapitola popisuje výsledky výkonnosti cache a plochy zabratej na FPGA (Virtex-II Pro). Na záver kapitoly je popísané ako bola overená funkčnosť výsledného návrhu.

4.1 Výkonnosť

Pre hodnotenie **latencie** je potrebné uvažovať dva prípady:

- Latencia v prípade ak je položka v cache nájdená, t.j. jedná sa o hit. V tomto prípade sa dáta zapíšu, resp. dáta sú k dispozícií za tri hodinové cykly. Vyplýva to z blokových schém cache (obrázky 3.1 a 3.10). Je dobré poznamenať, že samotný zápis požiadavky do vstupnej fronty sa do výsledku nezapočítava.

- Latencia v prípade ak sa položka nenájde, čiže ide o miss, sa zvyšuje o čas nutný na spracovanie miss požiadavky (miss penalty). Dá sa to vyjadriť vzorcom

$$L = 3 + t_o + L_P,$$

  kde t_o je čas potrebný na zápis obete do hlavnej pamäte a L_P je latencia tejto pamäte. Pritom ani zápis obete ani latencia vyčítania položky nemusí byť konštantná. Keďže cache využíva write back princíp, určená obeť musí byť zapísaná iba vtedy, keď vznikla nekonzistencia medzi dátami v cache a hlavnou pamäťou. Čas t_o je možné teda vyjadriť takto:

$$t_o = 1 + st_b,$$

  kde s nadobúda hodnotu jedna alebo nula podľa toho či je nutné blok do pamäte zapísať alebo nie. Hodnota t_b udáva čas zápisu bloku do pamäte (počet cyklov) a je závislá na veľkosti bloku a charakteristikách hlavnej pamäte.

Návrh podporuje zreťazené spracovanie. Implementácia umožňuje pri postupnosti hit požiadaviek spracovať každý hodinový cyklus jednu z nich. Jediné obmedzenie nastane pri zmene typu zo zápisu na čítanie. Vtedy je potrebné reťazec zastaviť a najprv vykonať zápis a hneď po tom je možné zreťazene pokračovať v spracovávaní. Za normálnych okolností je **časový cyklus cache** jeden hodinový takt, ale v prípade práve popísanom je cyklus dva takty. Ak nastane miss, je potrebné zastaviť akékoľvek ďalšie spracovanie požiadaviek a teda cyklus sa predĺži o čas nutný na spracovanie miss požiadavky.

Posledným faktorom je **priepustnosť**. V tomto prípade nie je možné konkrétne určiť hodnoty, pretože sa jedná o generický návrh a priepustnosť závisí na konkrétnych parametroch. Obecne však za určujúce je možné považovať tieto body:

- Šírka slova
- Frekvencia
- Podiel hit a miss požiadaviek, ktoré určujú ako dlho bude procesor zablokovaný

4.2 Plocha na čipe a frekvencia

Na porovnanie vlastností cache bola opakovane vykonaná syntéza, pričom vždy sa menil práve vybraný parameter a ostatné boli konštantné. Keďže cache má pomerne veľké množstvo parametrov, tento postup je nevyhnutný. Pri hodnotení výslednej implementácie je potrebné zistiť nasledujúce tri hodnoty:

1. Počet **slice** jednotiek.
2. Počet **BRAM** jednotiek. Spolu so slice udávajú celkovú zabratú plochu na čipe.
3. Maximálnu **frekvenciu**.

Príloha A obsahuje graf pre každý parameter, ktorý ma vplyv na výslednú plochu alebo frekvenciu. Na syntézu bol použitý nástroj "Precision RTL Synthesis".

4.2.1 Plocha

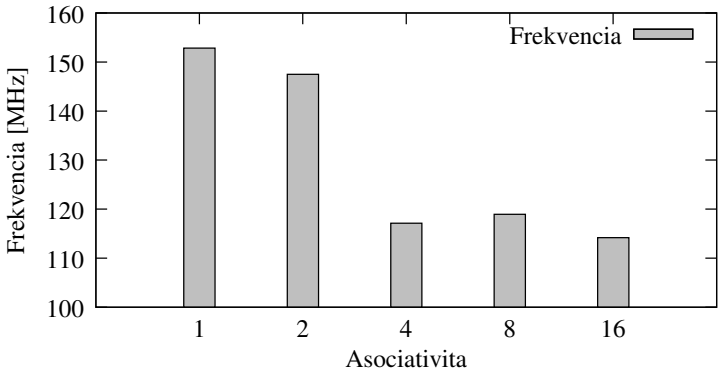
Výsledky syntézy ukazujú lineárnu závislosť počtu blokových pamätí na všetkých parametroch, okrem spôsobu výberu obete, ktorý prakticky na to nemá žiadny vplyv. Je to dané tým, že tieto parametre priamo ovplyvňujú výslednú kapacitu cache.

Pri počtoch slice jednotiek sa taktiež ukazuje, že závislosť na parametroch (opäť to neplatí pre výber obete) je lineárna. To by na prvý pohľad mohlo byť prekvapivé, pretože parametre ako veľkosť bloku, veľkosť cache, a dátová šírka priamo určujú jej kapacitu, a tá je daná počtom BRAM. Asociativita má v tomto prípade vplyv na počet vstupov multiplexoru, ktorý prepína medzi jednotlivými cestami v dvojrozmerných pamätiach. Presne to zobrazuje obrázok 3.5. Tento multiplexor je potrebný pri výstupe takmer zo všetkých týchto pamätí. Veľkosť cache má vplyv (tak isto ako aj pre BRAM) na počet položiek v distribuovaných pamätiach, v ktorých sú uložené informácie pre výber obete a dáta v zneplatňovacej pamäti. Veľkosť týchto pamätí rastie taktiež lineárne s počtom riadkov. Podobne ako v distribuovaných pamätiach, aj veľkosť blokových pamätí má nemalý vplyv na počet slice. Samotné výsledné pamäte (paralelne zobrazené na obrázku 3.5) totiž nie sú tvorené jedinou BRAM, ktorá má obmedzené množstvo položiek. Pri dátovej šírke väčšej ako 32 bitov maximálny počet položiek v jednej BRAM je 512 [12]. Ak je teda zvolený veľký počet riadkov – buď prostredníctvom veľkosti cache, alebo veľkosti bloku – môže sa stať že bude potrebné jednotlivé BRAM kaskádne spojovať. Na spojenie je nutné vyhradiť nejaké množstvo logiky a pri veľkých dátových šírkach môže práve toto kaskádne zapojenie zaberať väčšinu plochy na čipe.

Spôsob výberu obete, vzhľadom na predchádzajúce parametre, nijako výrazne neovplyvňuje počet slice. Za zmienku stojí iba náhodný výber, ktorý na rozdiel od PLRU a FIFO, nepotrebuje uskladňovať svoj stav pre všetky riadky, ale iba do jediného registru. Týmto sa ušetrí asi 100 slice pre danú konfiguráciu.

4.2.2 Frekvencia

Najväčší vplyv na frekvenciu má asociativita. Je to dané tým, že pri vzrastajúcej asociativite sa zvyšuje podiel kombinačnej logiky pri asociatívnom vyhľadávaní v cache. Okrem toho, so zväčšujúcim sa počtom ciest narastá aj počet vstupov multiplexorov vyberajúcich dáta zo zvolenej cesty v dvojrozmerných pamätiach. Ako vyplýva z grafu na obrázku 4.1, veľký skok – z priemerne 145 MHz na 115 MHz – nastáva pri prechode z 2-cestne asociatívnej



Obrázok 4.1: Závislosť frekvencie na asociativite cache. Kompletné grafy sú v prílohe A.

cache na 4-cestnú. Pri požiadavke vyšších frekvencií za nutnosti aspoň minimálnej asociativity je preto výhodné požiť 2-cestnú cache, ktorá úspešne konkuruje priamo mapovanej. Podstatne vyššie frekvencie pri priamo mapovanej a 2-cestnej cache je možné vysvetliť aj na základe týchto dvoch faktov:

1. Priamo mapovaná cache nevyžaduje žiadny výber obete ani výber voľnej cesty – dva prvky, ktoré do kombinačnej cesty vnášajú podstatné oneskorenie.

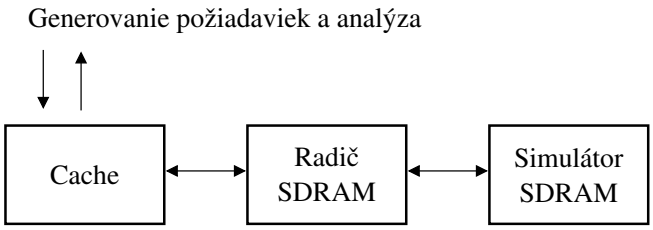
2. 2-cestná cache síce vyžaduje už výber obete a jednej z dvoch voľných ciest (preto má o niečo nižšiu frekvenciu) ale tento výber je veľmi jednoduchý a neprináša žiadne podstatné zaťaženie.

Ďalším z parametrov, ktorý výraznejšie ovplyvňuje frekvenciu, je počet riadkov (veľkosť cache). Zatiaľ čo menšie kapacity prakticky nemajú na výsledok vplyv, približne od 1024 položiek sa frekvencia začína približovať 100 MHz. V danej konfigurácii určujúcej výslednú kapacitu (radovo jeden až dva MB), je to stále pomerne dobrý výsledok.

Na rozdiel od dvoch predchádzajúcich parametrov, veľkosť bloku a algoritmus výberu obete majú na frekvenciu minimálny vplyv. Taktiež, ako by sa dalo predpokladať, frekvencia je na šírke slova nezávislá.

4.3 Overenie funkčnosti

Na funkčné overenie správnosti implementácie bolo vytvorené prostredie na odsimulovanie činnosti cache v programe Modelsim. Prostredie obsahuje simulačné komponenty dynamickej pamäte a jej radiča spolu s cache (obrázok 4.2). Ako základ slúži testbench (napísaný tak isto vo VHDL), ktorý je vytvorený generický, viazaný na parametre samotnej cache. Takto bolo možné dôkladne otestovať rôzne stavy a funkcionalitu výsledného produktu ešte predtým ako sa vôbec začne reálne používať.



Obrázok 4.2: Simulácia cache použitím modelu dynamickej pamäte a jej radiča.

Funkčnosť bola taktiež otestovaná v hardware, v projekte FlowMon monitorovacej sondy [13]. Cache bola umiestnená medzi *Storage procesor (STO)* a *pamäť NetFlow záznamov*. Následne bola generovaná sieťová prevádzka tak aby bolo možné sledovať jej funkčnosť.

Kapitola 5

Záver

Cieľom tejto práce bol návrh a implementácia cache pamäte konfigurovateľnej pomocou sady generických parametrov tak, aby ju bolo možné prispôsobiť širokému spektru rôznych aplikácií. Boli naštudované princípy konštrukcie cache pamätí a vykonaný dôkladný rozbor rôznych typov pamätí. Získané informácie sú zhrnuté v kapitole 2. Na základe získaných znalostí bol vytvorený návrh a generická implementácia v jazyku VHDL s ohľadom na technológiu Virtex-II Pro.

Navrhnutá komponenta je konfigurovateľná veľkým množstvom parametrov. Je možné definovať dátovú šírku, veľkosť bloku, asociativitu, ale aj spôsob výberu obete pri zaplnení cache. Boli implementované tri prístupy - Pseudo LRU, FIFO a náhodný výber. Všetky parametre majú vplyv na frekvenciu a použité zdroje v FPGA. Závislosť využitia FPGA a maximálnej dosiahnuteľnej frekvencie na jednotlivých parametroch je zhrnutá v kapitole 4. Pre zvýšenie priepustnosti bola tiež implementovaná podpora zreťazeného spracovania, vďaka ktorej je cache schopná spracovať v každom hodinovom cykle jednu požiadavku na čítanie alebo zápis. Vytvorená implementácia bola overená pomocou simulácií v programe Modelsim a testovaná na karte Combo6X.

Výsledná komponenta si nájde uplatnenie ako rýchla pamäť ku procesnej jednotke pri monitorovaní sietí, pri vyhľadávaní položiek v IP smerovačoch, poprípade ako cache pre aplikačne špecifické procesory pracujúce napríklad s dynamickou pamäťou. Samotné možnosti sú však natoľko flexibilné, že cache je možné použiť obecne v akýchkoľvek systémoch, ktoré sú schopné využiť princíp hierarchického členenia pamäte.

Aj keď predchádzajúci zoznam vymenoval značné množstvo flexibility, návrh nemusí nutne byť považovaný za uzavretý. Jedným z parametrov, ktoré v konečnom dôsledku určujú výkonnosť cache je jej latencia, ktorá je ale pre tento prípad konštantná. Ako rozšírenie do budúcnosti sa preto dá počítať s latenciou ako ďalším parametrom. Odstránením vstupnej a výstupnej fronty by totiž bolo možné ušetriť až dva hodinové cykly. Taktiež register v zreťazenom spracovaní by bol voliteľný, čo by ušetrilo ďalší cyklus. Tento prístup by mal oporu hlavne v miniatúrnych cache pamätiach, kde by bolo možné dosiahnuť latenciu až jeden hodinový cyklus. Samozrejme, iba v prípade hit požiadaviek. Na druhej strane s veľkou pravdepodobnosťou by takýto zásah mal výrazný negatívny vplyv na výslednú frekvenciu.

Literatúra

[1] Alpert, D. B.; Flynn, M. J.: Performance Trade-Offs for Microprocessor Cache Memories. *IEEE Micro*, 1988: s. 44–54.

[2] Denning, P. J.: The working set model for program behavior. *Commun. ACM*, ročník 11, č. 5, 1968: s. 323–333, ISSN 0001-0782, doi:http://doi.acm.org/10.1145/363095.363141.

[3] Ghasemzadeh, H.; Mazrouee, S. S.; Kakoee, M. R.: Modified Pseudo LRU Replacement Algorithm. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, Washington, DC, USA: IEEE Computer Society, 2006, ISBN 0-7695-2546-6, s. 368–376, doi:http://dx.doi.org/10.1109/ECBS.2006.52.

[4] Grossman, J.: A Systolic Array for Implementing LRU Replacement.
URL http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-18.pdf

[5] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture: A Quantitative Approach*. 2003.

[6] Jeong, J.; Dubois, M.: Cost-Sensitive Cache Replacement Algorithms. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1871-0, str. 327.

[7] Kroft, D.: Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, s. 81–87.

[8] Liu, H.; Weng, S.; Sun, W.: Cache, Matrix multiplication and Vector. http://www.cs.umd.edu/class/fall2001/cmsc411/proj01/cache/cache.html.

[9] Smith, A. J.: Cache Memories. *ACM Comput. Surv.*, ročník 14, č. 3, 1982: s. 473–530, ISSN 0360-0300, doi:http://doi.acm.org/10.1145/356887.356892.

[10] William Stallings: *Computer Organization and Architecture: Designing for Performance*. 2005.

[11] Talbot, B.; Sherwood, T.; Lin, B.: Cached IP Lookup for Terabit Speed Routers. In *Proceedings of the IEEE Global Communications Conference (GlobeCom)*, 1999.

[12] Xilinx: Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. http://direct.xilinx.com/bvdocs/publications/ds083.pdf.

[13] Žádník, M.; Pečenka, T.; Kořenek, J.: NetFlow Probe Intended for High-Speed Networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL05)*, IEEE Computer Society, 2005, ISBN 0-7803-9362-7, s. 695–698. URL http://www.fit.vutbr.cz/research/view_pub.php?id=7836

Príloha A

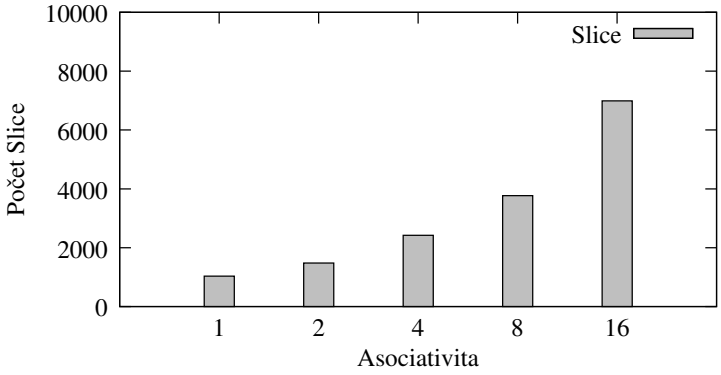
Grafy výsledkov syntézy

Táto príloha obsahuje podrobné výsledky syntézy použitím "Precision RTL Synthesis". Tabuľka A.1 zhrňuje konfigurácie počas jednotlivých pokusov. Pre každý riadok je definovaný jeden parameter, ktorý sa počas syntézy mení, pričom ostatné zostávajú konštantné. Na to nadväzujú grafy závislostí zabratej plochy a frekvencie na týchto meniacich sa hodnotách.

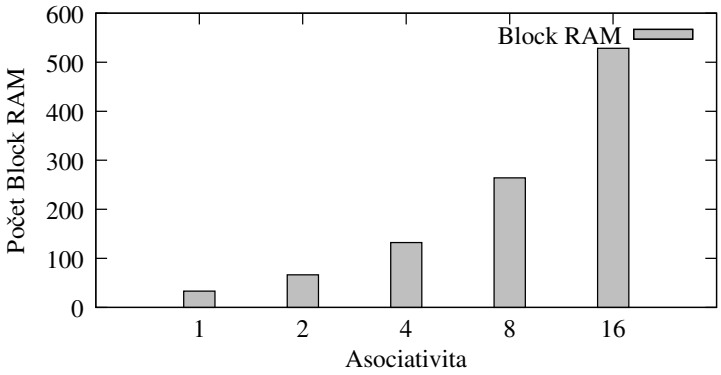
Parameter	Asociativita	Veľkosť cache [počet riadkov]	Veľkosť bloku [počet slov]
Asociativita	1 – 16	1024	4
Veľkosť cache	4	64 – 4096	4
Veľkosť bloku	4	512	1 – 16
Výber obete	4	1024	4
Šírka slova	4	512	4

Parameter	Výber obete	Šírka slova [b]	Kapacita [B]
Asociativita	FIFO	128	65k – 1M
Veľkosť cache	PLRU	128	16k – 1M
Veľkosť bloku	PLRU	128	32k – 0.5M
Výber obete	všetky	128	256k
Šírka slova	PLRU	8 – 256	8k – 128k

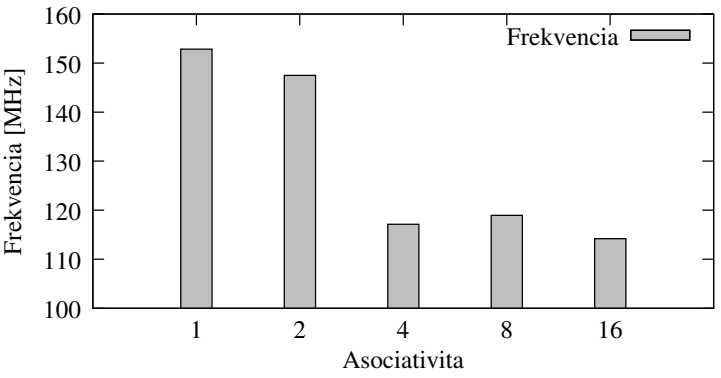
Tabuľka A.1: Nastavenie parametrov pre testovanie výsledkov syntézy.



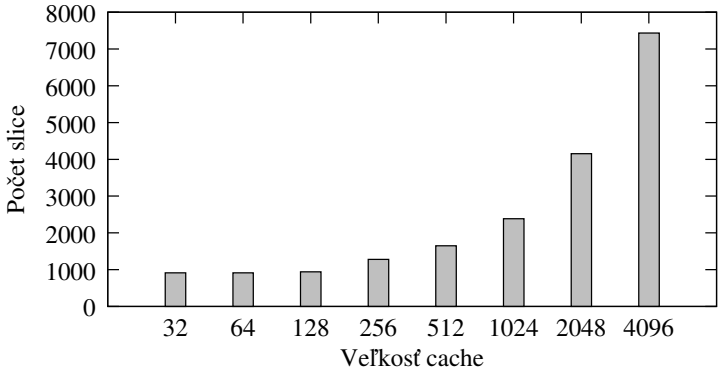
Obrázok A.1: Počet slice v závislosti od asociativity.



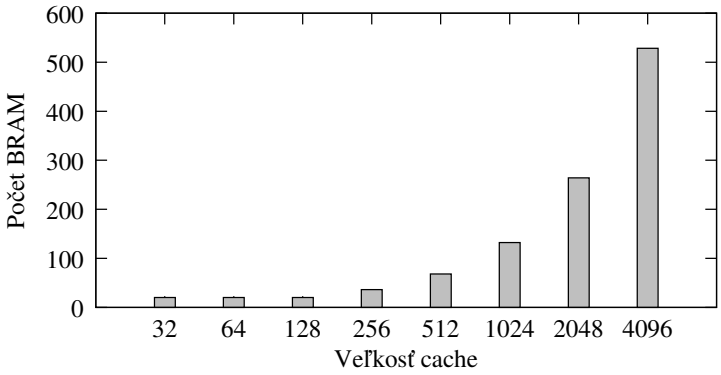
Obrázok A.2: Počet BRAM v závislosti od asociativity.



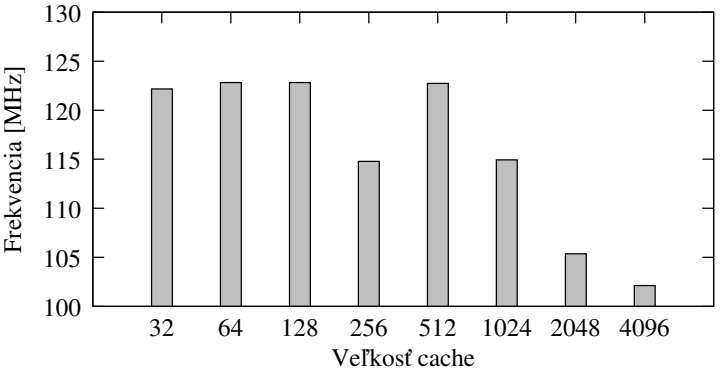
Obrázok A.3: Frekvencia po syntéze v závislosti na asociativite.



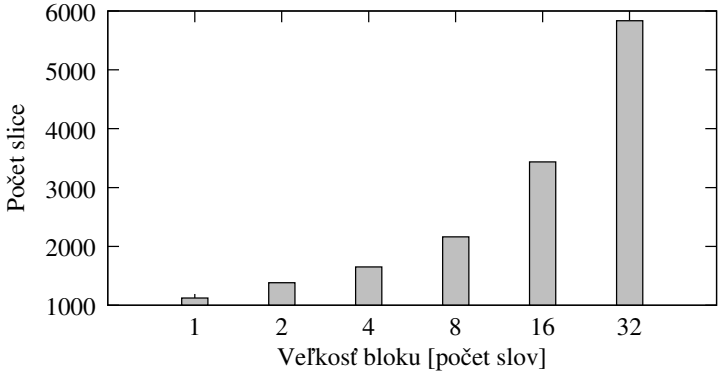
Obrázok A.4: Počet slice v závislosti na veľkosti cache.



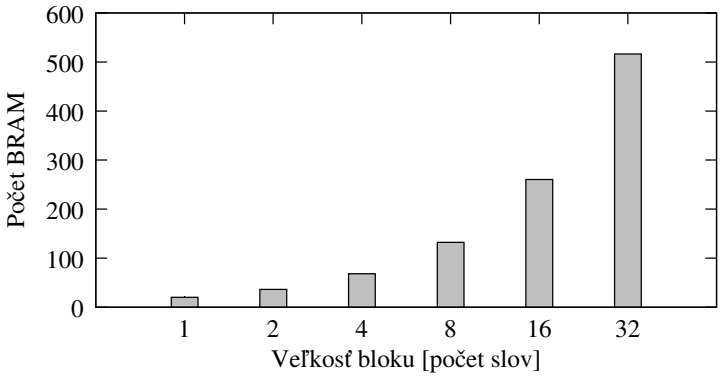
Obrázok A.5: Počet BRAM v závislosti na veľkosti cache.



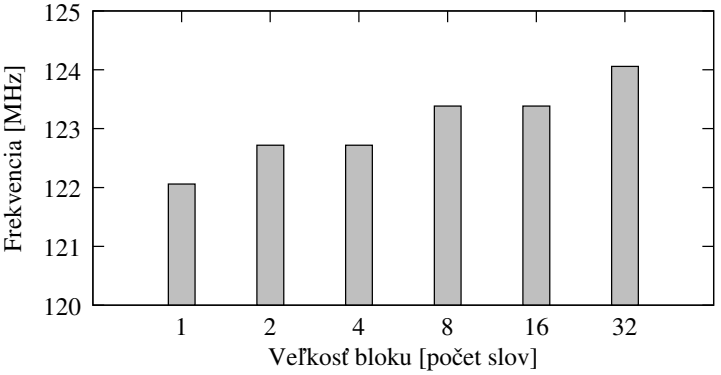
Obrázok A.6: Frekvencia po syntéze v závislosti na veľkosti cache.



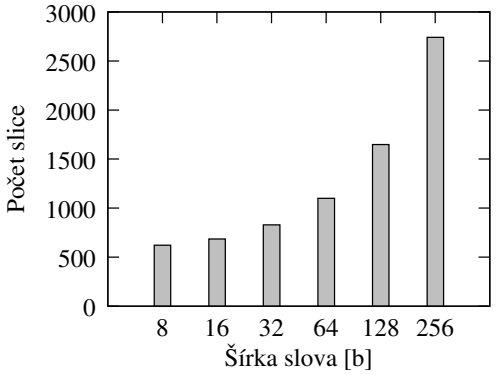
Obrázok A.7: Počet slice v závislosti na veľkosti bloku.



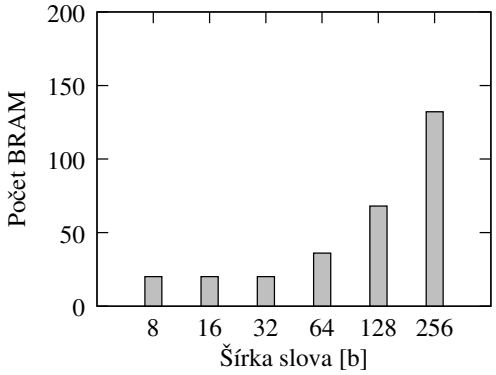
Obrázok A.8: Počet BRAM v závislosti na veľkosti bloku.



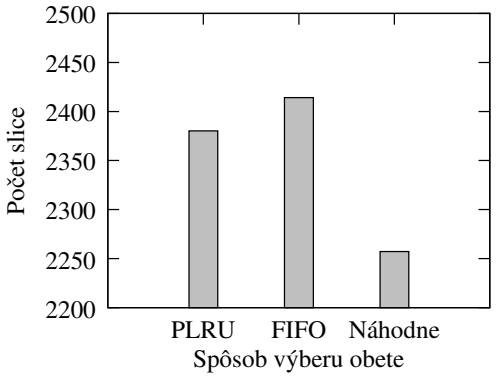
Obrázok A.9: Frekvencia po syntéze v závislosti na veľkosti bloku.



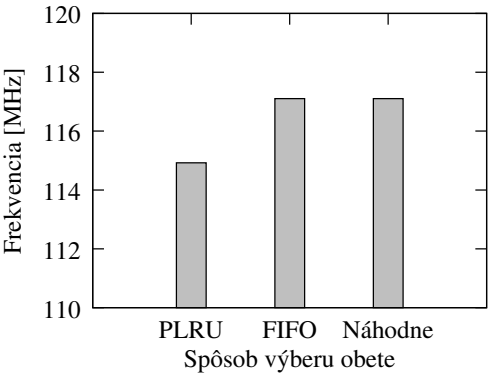
Obrázok A.10: Počet slice v závislosti na šírke slova.



Obrázok A.11: Počet BRAM v závislosti na šírke slova.



Obrázok A.12: Počet slice v závislosti na spôsobe výberu obete.



Obrázok A.13: Frekvencia v závislosti na spôsobe výberu obete.

Príloha B

CDROM

K práci je priložené CD, ktoré obsahuje nasledujúce súčasti:

- Zdrojový text tejto práce so všetkými náležitosťami, tak aby ju bolo možné znovu vytlačiť,
- programovú dokumentáciu,
- zdrojové texty výslednej cache pamäte.