VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

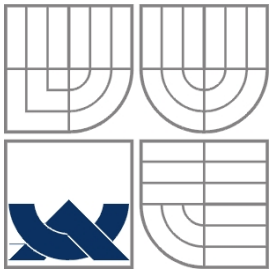
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH POKROČILÉ ARCHITEKTURY PROCESORU V JAZYCE VHDL

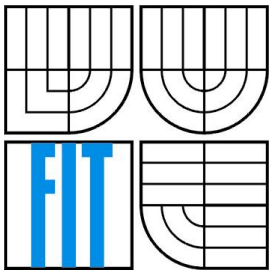
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Daniel Slavík
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH POKROČILÉ ARCHITEKTURY PROCESORU V JAZYCE VHDL

VHDL DESIGN OF ADVANCED CPU

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE — Bc. Daniel Slavík
AUTHOR

VEDOUCÍ PRÁCE — Ing. Martin Straka
SUPERVISOR

Abstrakt

Cílem projektu bylo prostudovat zřetězené architektury procesorů, dále pak architektury instrukčních a datových cache. Vybraná zřetězená architektura měla být navržena včetně instrukční a datové cache a implementována v jazyce VHDL. Projekt jsem pojal tak, že jsem implementoval nejprve subskalární architekturu, poté tři verze skalární architektury. Byla provedena syntéza těchto architektur do FPGA a na zvoleném algoritmu porovnána jejich výkonnost. V další části práce jsem navrhl a implementoval instrukční i datovou cache pro obě architektury. Tyto cache se mi však už nepodařilo syntetizovat. Závěrečná kapitola této práce pojednává o superskalární architektuře, což je architektura používaná v dnešní době.

Abstract

The goal of this project was to study pipelined processor architectures along with instruction and data cache. Chosen pipelined architecture should be designed and implemented using VHDL language. Firtsly, I decided to implement the subscalar architecture first, secondly, three versions of scalar architecture. For these architectures synthesis into FPGA was done and performance of these architectures was compared on chosen algorithm. In the next part of this thesis I designed and implemented instruction and data cache logic for both architectures. However I was not able to synthetise these caches. Last chapture of this thesis deals with the superscalar architecture, which is the architecture of nowadays.

Klíčová slova

procesor, VHDL, subskalární, skalární, superskalární, cache, RAM, návrh

Keywords

processor, VHDL, subscalar, scalar, superscalar, cache, RAM, design

Návrh pokročilé architektury v jazyce VHDL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením ing. Martina Straky

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Daniel Slavík
25.5. 2010

Poděkování

Poděkování ing. Martinu Strakovi za konzultace a odbornou pomoc.

Obsah

1 Úvod

Cílem diplomové práce je navrhnout a implementovat zřetězenou architekturu procesoru včetně logiky instrukční a datové cache. Implementovanou architekturu se pokusit syntetizovat do FPGA a diskutovat dosažené výsledky.

Práce je v zásadě rozdělena do dvou částí. První část se zabývá architekturou, vlastnostmi a návrhem pamětí cache. Druhá část se zabývá architekturami procesorů. Popsány jsou subskalární, skalární a superskalární architektury, implementovány jsou první dvě. Po krátkém úvodu následuje kapitola zabývající se paměťmi cache. V teoretické části je popsáno, proč tyto paměti vůbec vznikly, jsou rozebrány jejich vlastnosti, architektura a konečně vlastní návrh. Třetí kapitola stručně popisuje jazyk VHDL. Tato kapitola není učebnicí tohoto jazyka, popsány jsou zejména konstrukce, které byly použity při programování tohoto projektu. Ve čtvrté kapitole je popsána instrukční sada, která je implementována všemi verzemi procesorů. Nejdůležitější kapitolou je kapitola probírající samotné architektury procesorů. Po teoretickém úvodu následuje vždy podrobný popis jednotlivých částí procesoru na diagramech z prostředí ModelSim je demonstrována funkčnost na několika vybraných algoritmech. Závěr shrnuje poznatky získané při zpracování této práce.

1.1 Historický vývoj a rodělení architektur

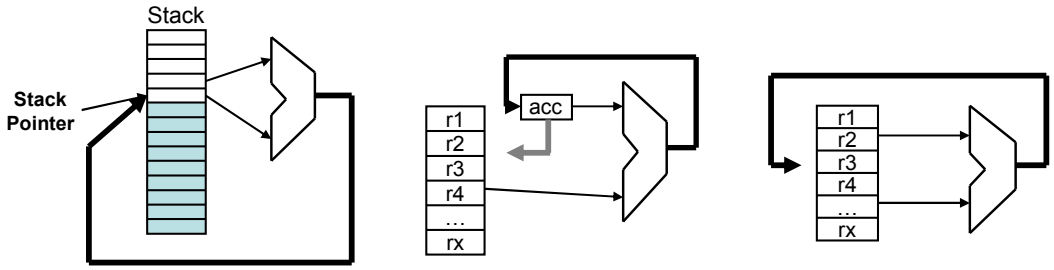
Historie počítačů sahá do první poloviny 20. století. Na počátku byl program, který procesor vykonával, pevnou součástí hardwaru a změna programu znamenala výměnu hardwaru. To v některých případech (malé mikrokontroléry) přetrvává do dnes. Tento způsob je používaný např. základní verzí Turingova stroje, kdy páska obsahuje pouze data a podle pozice čtecí hlavy a obsahu pásky se rozhoduje o dalším výpočetním kroku. Universální Turingův stroj už však na pásce obsahuje i samotný kód programu.

Byly prezentovány dvě významné architektury procesorů. **Von Neumannova architektura** byla prezentována v roce 1945 Johnem von Neumannem. Její myšlenkou je mít jednu společnou paměť, ve které budou uložena jak data tak instrukce programu. Umožňuje tedy sekvenčně načítat buď data nebo instrukce programu. Oproti tomu byla uvedena tzv. **Harwardská architektura** prezentovaná na Harwardské universitě. Paměť pro instrukce i data je oddělená. Instrukce byly původně uloženy na děrovací pásce a data v jakémsi mechanickém čítači. Tato architektura umožňuje principiálně paralelní přístup k datům a instrukcím. Současné procesory jsou spíše kombinací obou těchto architektur. Data i instrukce jsou sice umístěné společně v jedné paměti RAM, z této paměti jsou však načteny do oddělených pamětí cache. Jedná se většinou o cache úrovně L1.

Jiné možné historické rozdělení procesorů je:

- zásobníková architektura
- střadačová architektura
- registrová architektura

Schémata ilustrující princip těchto architektur znázorňuje obrázek Chyba! Nenalezen zdroj odkazů.. Zásobníková architektura je význačná tím, že je paměť organizovaná jako zásobník, jako vstupní operandy mohou být použity tedy pouze ty z vrcholu zásobníku. Střadačová architektura se vyznačuje tím, že má jeden zvláštní registr – střadač, ze kterého vždy načítá jeden operand a zároveň do něj ukládá výsledek. Registrová architektura je architektura používaná v současnosti. Zdrojové operandy jsou načítány z libovolného z registrů, výsledek je též ukládán na libovolné místo [1].



Zásobníková architektura Střadačová architektura Registrová architektura

Obr. 1 – Historický vývoj architektur procesoru

Procesory je dále možné rozdělit podle svého účelu. Cílem diplomové práce je pracovat s universálními procesory, které umožňují zpracovat libovolnou úlohu všech ostatních typů procesorů. Mimo ně však existují nejrůznější specializované procesory pro konkrétní účel – grafické, síťové, DSP, akcelerátory nejrůznějších procesů, mikrokontroléry pro vestavěná zařízení atd.

Tabulka 1 shrnuje vývoj universálních procesorů pro PC.

Typ architektury	Počet vydávaných instrukcí do funkčních jednotek za takt	Provádění instrukcí	IPC
Subskalární	0-1	Sekvenční	<< 1
Skalární	0-1	Paralelní – časový paralelismus	< 1
Superskalární	0-m (n > m)	Paralelní – časový + prostorový paralelismus	< m
Vícevláknová	0-n (o > n)	Paralelní – časový + prostorový paralelismus	< n
Vícevláknová + vícejádrová	0 - o	Paralelní – časový + prostorový paralelismus	< o

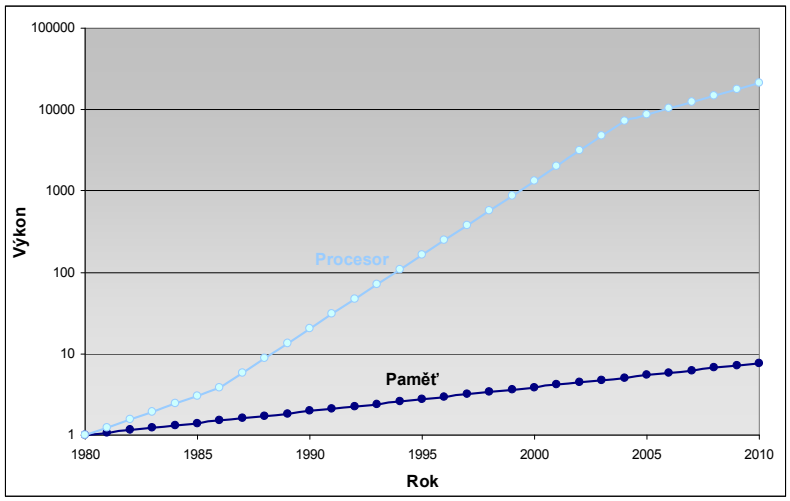
Tabulka 1 – Přehled a stručný popis architektur universálních procesorů

CPI…Clock Per Instruction – počet hodinových taktů potřebných pro zpracování jedné instrukce

IPC…Instruction Per Cycle – je počet instrukcí, které je možné zpracovat během jednoho taktu. Je to vlastně inverzní parametr k CPI.

2 Cache a paměti RAM

Pro maximální výkonnost je důležité, aby rychlost operační paměti byla stejná jako rychlost procesoru. Tedy ideálně abychom měli dostatečně velkou a rychlou paměť do níž by přístup trval pouze jeden takt. Od počátku vývoje však rychlost procesorů roste rychleji než rychlost pamětí. Statisticky rychlost pamětí roste přibližně 1,07x za rok oproti růstu rychlosti procesoru 1,3 za rok. Pomyslné nůžky mezi rychlostí procesoru a pamětí RAM se tedy stále více rozvírají. Tuto skutečnost ilustruje obrázek 2.



Obr. 2 – srovnání rychlosti CPU a hlavní paměti

Řešením této situace je vytvoření hierarchie pamětí. Každá paměť, blíže k procesoru bude mít menší kapacitu a bude rychlejší. Data z paměti vyšší úrovně jsou mapována do paměti nižší úrovně. Při zpracování kódu programu tedy nejsou data načítána přímo z hlavní paměti do registrů procesoru. Místo toho jsou nejprve načtena do paměti cache a teprve odsud do registrů. Tento postup funguje díky tzv. principu lokality. Procesor obvykle stráví 90% času prováděním pouze 10% instrukcí – **časová lokalita** (temporal locality). Dále bylo zjištěno, že pokud procesor přistupuje k datům na určité adrese, je pravděpodobné, že bude přistupovat i k okolním adresám. – **prostorová lokalita** (spatial locality). Je tedy snaha o minimalizaci počtu přístupů do hlavní paměti.

Rychlost přístupu do paměti je možné charakterizovat pomocí přístupových dob:

Access time – udává dobu mezi tím kdy paměť obdržela požadavek na čtení a tím kdy byla data dodána.

Cycle time – udává za jakou dobu je možné vystavit následující adresu (je totiž potřeba určitý čas pro ustálení adresy)

Paměti cache jsou statické (SRAM). Jsou tvořeny poli klopných obvodů (R-S), pro uchování jednoho bitu je obvykle použito 6 tranzistorů. Pro uchování hodnoty je potřeba minimální energie. Cycle time přibližně odpovídá access time.

Paměti RAM jsou dynamické (DRAM). Pro uchování jednoho bitu je obvykle použit pouze jeden tranzistor a přečtení bitu znamená znehodnocení uložené hodnoty. Proto musí být hodnota bitu po každém čtení obnovena. Díky tomu je cycle time o hodně větší než access time. Aby se předešlo ztrátě musí být navíc data v RAM pravidelně obnovována, což znamená, že v tuto chvíli není možné z paměti číst. Pokud tedy v tento okamžik nastane výpadek z cache, bude jeho obsloužení trvat daleko déle.

Se zvyšováním kapacity se také zvětšovala šířka adresy do RAM. Postupem času přerostl počet vstupních pinů únosnou hodnotu a bylo rozhodnuto, že se adresa bude

zasílat ve dvou částech. Paměť RAM je koncipována jako matice. Proto je první část adresy označována jako RAS (Row Access Strobe) – adresa řádku. Druhá část adresy je označována jako CAS (Column Access Strobe) – adresa sloupce. Obnova dat v RAM probíhá přečtením obnovované části. Vystavením adresy CAS je možné provést v jeden okamžik obnovu celého sloupce. Proto je možné říct, že doba obnovy celé RAM je přímo úměrná druhé odmocnině její kapacity.

Určité vylepšení přístupu do RAM přineslo bufferování načítaného řádku. Řádek, ze kterého je položka přečtena je uložen do bufferu. Pokud je následující přístup opět do stejného řádku, nemusí se už posílat adresa RAS. Této technice se říká **fast page mode** [8].

2.1 Organizace paměti cache

Při žádosti o čtení z paměti se díky znalosti prostorové lokality a vysoké propustnosti RAM nenačítá pouze požadovaný operand, ale celý blok dat určité velikosti. Tento blok odpovídá velikosti bloku v cache. Velikost bloku je závislá na architektuře a liší se procesor od procesoru. Typická hodnota pro x86 architekturu je 64B. Cache je dále členěna na skupiny bloků (*sets*). Všechny skupiny obsahují stejný počet bloků. Extrémní případy jsou když má cache pouze jednu skupinu nebo naopak skupina má jenom jeden blok. Při načítání bloků z operační paměti jsou tyto bloky algoritmem nejprve mapovány na konkrétní skupinu, v rámci této skupiny jsou pak umístěny na některé volné místo [8]. Podle počtu skupin v cache je možné vytvořit rozdělení:

Jednocestná asociativní cache (Přímé mapování)

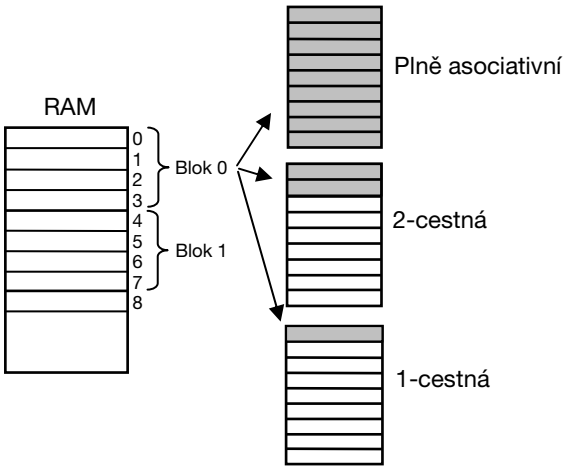
Každá skupina bloků v cache obsahuje pouze jediný blok. Blok z operační paměti je tedy vždy mapován na jedno stejné místo. Jako mapovací funkci je možné použít

*(Adresa bloku v RAM) mod (Počet **bloků** v cache)*

N-cestná asociativní cache

Skupina v cache obsahuje *n* bloků. Po namapování bloku na skupinu se rozhoduje kam umístit blok v rámci této skupiny. Konkrétní blok z RAM je vždy namapován vždy na stejnou skupinu, v rámci skupiny však může mít pokaždé jiné umístěni. Jako mapovací funkci je možné použít

*(Adresa bloku v RAM) mod (Počet **skupin** v cache)*



Obr. 3 – Paměti cache

Plně asociativní cache

Cache má pouze jedinou skupinu – konkrétní blok z operační paměti tedy může být namapován kamkoli.

Rozdělení cache podle velikosti skupin ilustruje obr. 3. Nultý blok z paměti RAM je mapován na nultou skupinu v cache, šedě jsou pak znázorněny bloky, do kterých může být načtený blok umístěn. Pokud je ve skupině volné místo, může být blok umístěn libovolně v rámci tohoto místa. Pokud však v rámci skupiny není volné místo, je potřeba odsunout některý z bloků do paměti vyšší úrovně.

S narůstající rychlostí procesorů vznikla potřeba vytvořit další úroveň (L2) paměti cache. Tato paměť je větší než L1 cache, má ale větší přístupovou dobu. Obecně platí pravidlo, že při požadavku na čtení dat z RAM jsou data nejprve načtena do L2 cache a teprve odsud do L1. Toto se opět liší u různých architektur a různých výrobců, stejně tak i stupeň asociativity.

Procesor AMD Athlon 64 X2 používá L1 cache o velikosti 2x64KB (cache pro instrukce a pro data), L2 cache o velikostech 256/512/1024KB. L2 cache je 16ti cestná asociativní. Každé jádro má svou L2. Při výpadku z L1 je zkontrolováno jestli se data nenachází v L2. Pokud ani tam nejsou, jsou data načtena z paměti RAM přímo do L1. Data tedy nejsou redundantně umístěna v L1 i v L2 současně. Do L2 se můžou data dostat dvojím způsobem. Pomocí hardwarového dopředného načtení nebo odsunem bloku dat z L1. Softwafové dopředné načítání způsobí, že jsou data načtena přímo do L1. SW dopředné načítání má dále možnost označit blok speciálním tagem aby nemohl být odsunut do L2. Tímto se dá zabránit „znečištění" L2 cache. Tato technika se označuje jako *non-temporal prefetch.* AMD má dále implementovanou techniku Streaming Store umožňující pomocí zvláštní instrukce zapisovat data do RAM bez toho aniž by bylo potřeba blok z RAM nejprve načítat do cache. Toto je výhodné pokud programátor ví, že daný blok nebude v budoucnu potřebovat [11].

2.2 Odsunutí bloku z cache

Jsou v zásadě tři základní možnosti, který blok ze skupiny odsunout.

Vybrat náhodný blok

Toto řešení je beze sporu nejjednodušší a taky nejsnáze implementovatelné, ale neuvažuje jak často jsou bloky v cache využívány.

FIFO

Skupiny cache je možné organizovat jako paměť typu FIFO. Odsunut bude vždy nejdříve zapsaný blok.

LRU – Least Recently Used

Je nejspravedlivější variantou, odsunut je vždy ten blok, který je v rámci dané skupiny nejméně využíván. Toto řešení je ale zároveň nejsložitější na implementaci, u každého bloku je potřeba udržovat počet přístupů.

2.3 Čtení z paměti cache

Ideální případ je když se veškerá data programu vejdou do cache. Load/store operace pak prakticky nebrzdí běh programu. Při načítání dat se nejprve zkontroluje jestli je příslušný blok v cache. Pokud ano, mohou být data okamžitě načtena. Pokud ne, nastává výpadek bloku cache a data musí být načtena z paměti vyšší úrovně. Důvodů výpadku může být několik.

- Jedná se o **první načtení dat z daného bloku** v průběhu programu (*nucené* výpadky). Blok tak logicky nemůže být ještě přítomen v cache. Procento těchto výpadků je však velice malé.
- Výpadek nastal z **kapacitních důvodů**. Toto je odvislé od velikosti paměti cache. Pokud potřebuji načítat velké množství dat, cache se brzy zaplní a starší bloky musí být odsunuty. Řešení tohoto problému je zvěšení paměti cache nebo zavedení další úrovně. Na druhou stranu cache není možné zvětšovat libovolně, musí být dostatečně malá, aby se vešla k jádru procesoru a aby byla pokud možno co nejrychlejší.
- Výpadek z důvodu **konfliktu**. Výpadek v důsledku toho, že cache není plně asociativní. Pokud uvažujeme jednocestnou asociativní cache a načítané bloky se mapují stále na stejné místo, dochází s každým čtením k výpadku a odsunutí aktuálního bloku. Řešení tohoto problému je zvýšit stupeň asociativity ideálně na plně asociativní. Zvýšením asociativity však přibývá počet bloků, které musí být při každém čtení prohledány a tedy ke zpomalení přístupu do cache.

- Výpadek z důvodu udržování **koherence** cache. Tento druh výpadku se týká pouze multiprocesorů. Pokud mají dva procesory načtený do své cache stejný blok dat a jeden z nich zapíše do bloku data, musí si druhý procesor tento blok zneplatnit a při následujícím přístupu do něj načíst aktuální kopii z paměti vyšší úrovně. Tomuto jevu se říká *falešné sdílení*.

2.4 Zápis do paměti cache

Instrukce zápisu nejsou tak časté jako instrukce čtení z paměti. Výsledky operací jsou často předávány v registrech. Existují dvě strategie zápisu do paměti cache.

Write back – operand je zapsán pouze do paměti cache, ne do operační paměti. Do operační paměti (nebo obecně do paměti vyšší úrovně) je zapsán až v okamžiku, kdy je nutné blok z cache odsunout. Při použití této metody je jediná platná kopie v cache. V operační paměti je starý výsledek. Aby bylo jasné, že platná kopie je pouze v cache, je modifikovaný blok označen speciálním bitem – Dirty bit. Tato varianta šetří provoz na sběrnici mezi operační pamětí a cache.

Write through – operand je zapsán jak do operační paměti, tak do paměti cache. Tato varianta je náročnější na zatížení paměťové sběrnice a obecně pomalejší.

Obě varianty je možné ještě vylepšit zápisovými buffery. Pokud nastane výpadek bloku z cache, tak při načítání musí procesor čekat, dokud nebudou tyto operandy načteny – jsou totiž nutné pro provádění aktuální instrukce. Při zápisu je ale situace jiná. Procesor nepotřebuje v daný okamžik zapisovanou hodnotu. Data se proto ani při výpadku nemusí zapisovat okamžitě do operační paměti, ale je možné je zapsat do zápisového bufferu. Teprve odtud je možné je zapsat do paměti, až nastane vhodná chvíle.

2.5 Výpadek při zápisu do cache

Existují dvě možnosti, jak se zachovat, když se má zapsat výsledek do bloku, který není v cache.

Write allocate – blok, do kterého se má zápis uskutečnit, je načten do cache a do něj je pak proveden zápis. Této variantě se také někdy říká *read before write*.

No-write allocate – blok, do kterého se má zápis uskutečnit, není do cache načítán a výsledek se zapíše rovnou do operační paměti (případně nejprve do zápisového bufferu a teprve odtud do paměti).

2.6 Parametry cache a optimalizace

Správný návrh cache je klíčový pro výkonnost procesoru. Při návrhu je nutné zvolit správnou kombinaci různých parametrů cache. Ty nejdůležitější jsou:

Velikost cache – větší cache může snížit rizika kapacitních výpadků, zároveň však může zpomalit přístup do cache.

Velikost bloku – větší blok může opět snížit riziko nucených výpadků, zároveň však zvyšuje penalizaci za výpadek a roste-li velikost bloku neúměrně ku velikosti cache, riziko výpadku se naopak zvyšuje (závisí na prostorové lokalitě dat).

Stupeň asociativity – pokud bychom sledovali pouze četnost výpadků, tak by byla nejvhodnější plně asociativní cache, tedy cache obsahující pouze jedinou skupinu bloků.

Tím však narůstá složitost, a tedy i přístupová doba a cena. V praxi se většinou používají 4-/8-cestné cache.

Hierarchie pamětí – tato myšlenka se v praxi používá už dlouhou dobu. Idea je mít hierarchii pamětí, kde paměť vyšší úrovně je o něco pomalejší než aktuální úrovně, ale má vyšší kapacitu. V současnosti se nejvíce používají 3 úrovně cache.

Priorita výpadků – jak již bylo napsáno dříve, zapisované operandy většinou nejsou okamžitě potřeba, proto je dobré nastavit vyšší prioritu pro obsluhu výpadků čtení. Pro zápis je pak možné implementovat zápisový buffer.

Miss rate – jedna z možností, jak charakterizovat cache. Počítá se jako:

počet výpadků / počet přístupů do paměti

Počet výpadků na 1000 instrukcí – další z možností, jak charakterizovat cache

*(počet instrukcí s výpadkem / počet všech instrukcí) * 1000*

Průměrná doba přístupu do paměti

*doba přístupu do cache + % výpadků * doba přístupu do RAM*

Neblokující cache – vylepšení spočívá v tom, že výpadek z cache nezpůsobí znemožnění dalších přístupů do cache. Cache, která dokáže obsloužit další operaci čtení nebo zápisu, zatím co se obsluhuje výpadek, se označuje jako hit under 1 miss. Existují další vylepšení tohoto přístupu - hit under 2 misses, hit under 4 misses…

Hardwarové dopředné načítání – procesor, resp. cache, si může udržovat záznamy přístupů do paměti a snažit se v těchto přístupech najít vzor. Např. pokud se cyklicky prochází dlouhé pole. Procesor pak může rozhodnout o načtení těchto dat dříve než přijde samotná instrukce, která k nim přistupuje.

Softwarové dopředné načítání – je obdoba HW načítání popsaného výše. O dopředném načtení operandu však musí rozhodnout programátor speciální instrukcí (i zde tedy musí být HW podpora). Je však otázka, kam přesně tuto instrukci umístit, aby byla data načtena včas. Instrukce nesmí být umístěna ani moc dopředu (což by mohlo způsobit odsunutí jiného bloku, který je stále potřeba nebo potencionálně i odsunutí bloku, který jsem právě načetl jinou instrukcí) ani moc pozdě (data by nebyla připravena včas). Pro toto existují speciální manuály ke každému procesoru.

Critical word first – pokud nastane výpadek, je požadovaný operand načten nejprve do registru a teprv pak je načten zbytek bloku do paměti cache.

Early restart – v případě výpadku se začne požadovaný blok načítat normálně do cache. V okamžiku, kdy je k dispozici požadovaný operand, je okamžitě načten do registru. Nečeká se tedy na načtení celého bloku do cache.

Way prediction – tato technika se používá u dvou a vícecestných cache. V rámci každé skupiny bloků je predikováno, do kterého bloku se bude přistupovat příště. Ušetří se tedy čas strávený prohledáváním skupiny bloků. Pokud predikce nebyla správná, musí se skupina prohledat. Tato technika umožní v případě správné predikci stejně rychlý přístup do n cestné asociativní cache jako do přímo mapované cache.

Trace cache – technika použitá v Pentiu 4. Cache si udržuje jakousi posloupnost provedených instrukcí a podle toho umísťuje data po sobě do cache. Data nejsou tedy mapována tak, jak jsou umístěna za sebou v bloku, ale je snaha umísťovat je tak, jak budou postupně potřeba při běhu programu. Pro instrukční cache to tedy znamená umísťovat instrukce v tom pořadí, v jakém budou prováděny za sebou. Cache tedy musí spolupracovat s prediktory skoků. [8], [10]

2.7 Návrh instrukční a datové cache a paměti RAM

2.7.1 Instrukční cache

Instrukční cache je navržena jako dvoucestná asociativní cache s asynchroním čtením a synchronním zápisem bloku instrukcí z paměti RAM. Je v ní implementován algoritmus LRU řešící odstranění méně užívaného bloku, pokud je potřeba na dané místo nahrát jiný blok instrukcí. Při návrhu cache jsem se rozhodl pro tyto parametry:

- velikost cache - 32 instrukcí po 16b
- velikost bloku - 4 x 2B
- počet setů - 4
- počet bloku v setu – 2
- obsluha výpadků – 2 takty

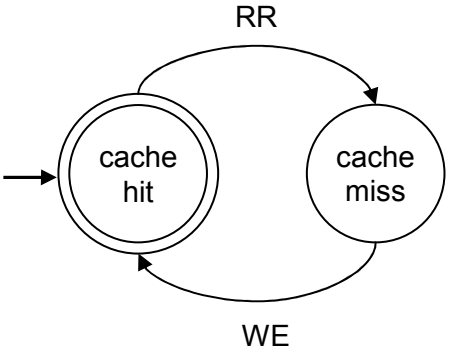
Cache je tedy organizována do 4 setů. Každý set obsahuje 2 bloky, každý blok obsahuje 4 instrukce. Blok dat z paměti RAM se mapuje na konkrétní set. V rámci setu může být umístěn libovolně. Mapování bloku z paměti RAM do cache probíhá pomocí následujícího vztahu:

adresa_bloku_v_RAM mod počet_setů_v_cache

Adresa bloku v ram je vpočítána pomocí následujícího vztahu:

adresa_instrukce div 4

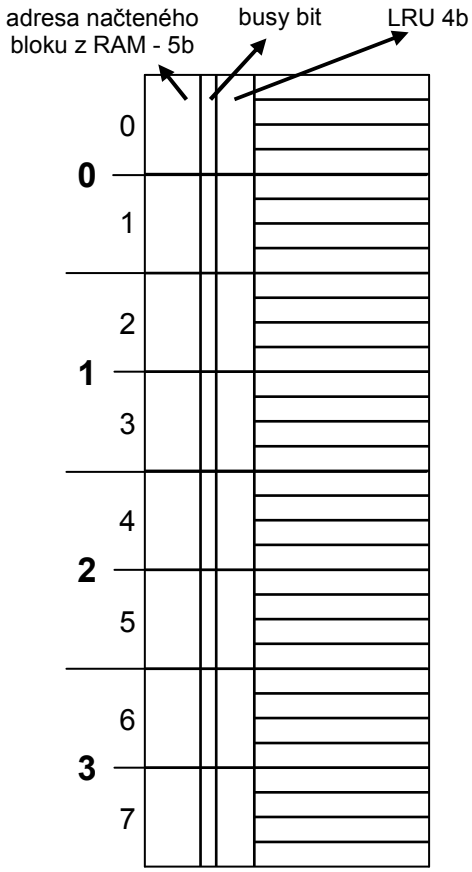
Princip činnosti je možné demonstrovat na konečném automatu zobrazeném na obrázku 4. Instrukční cache je v architektuře procesoru zařazena za programovým čítačem a na vstupu dostává adresu načítané instrukce z paměti RAM. Nejprve je tedy potřeba zjistit v jakém bloku cache se načítaná instrukce nachází. Blok paměti RAM je ekvivalentní s blokem cache (obsahuje 4 položky). Nejprve je tedy potřeba zjistit v jakém bloku v RAM se instrukce nachází. Blok RAM je mapován na konkrétní set paměti cache. Protože ale set obsahuje dva bloky, musí být oba tyto bloky prohledány. Pokud je v jednom z těchto bloků načten požadovaný blok RAM, jsou data asynchronně vystavena na výstup a pro daný blok je inkrementován čítač přístupů k němu. Vnitřní stav IC ve kterém jsou data načtena se nazývá **cache hit**. Pokud se však v daném setu nenachází požadovaný blok nastává výpadek z instrukční cache a vnitřní stav je změněn na **cache miss**. Do paměti RAM je zaslán signál `RR` (Read Request) a adresa načítané instrukce. Instrukční cache hledá v cílovém setu místo, do kterého bude načtený blok RAM umístěn. Pokud není volné místo ani v jednom bloku, musí být jeden blok nahrazen. V takovém případě je nahrazen



Obr. 4 – Stavový automat instrukční cache

blok s menší četností přístupů. Pokud mají oba bloky stejnou četnost přístupů, je nahrazen první z nich.

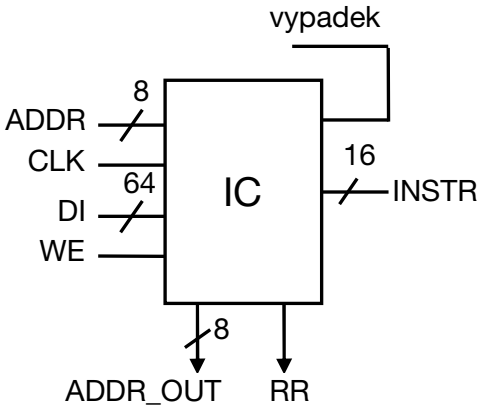
Paměť RAM obdrží signál RR a adresu načítané instrukce. S nástupnou hranou CLK vystaví data na výstup a nastaví signál `WE` označující, že jsou data připravena. S následující nástupnou hranou CLK zapíše IC tyto data, zruší nastavení signálu `RR`, pro dotčený blok vynuluje čítač přístupů a nastaví busy bit. Dále změní vnitřní stav na **cache hit** a vystaví data na výstup. Tímto je obsloužen výpadek.



Obr. 5 – Vnitřní struktura IC

Organizace paměti cache do bloků a setů je vidět na obrázku 5. Paměť je implementovaná jako pole signálů o délce 32 prvků typu `std_logic_vector`. Pro uložení metadat ke pro jednotlivé bloky jsou použity další tři pole. Pole o šířce 5b pro uložení adresy bloku RAM (protože se na jeden set mapuji více než dva bloky RAM, musím mít u každého bloku cache uloženou informaci o tom, ktarý blok RAM je v něm načten). Busy bit u každého bloku udává jestli je daný blok prázdný nebo v něm jsou načteny data z RAM. Každý blok navíc obsahuje čítač přístupů. Pro toto pole jsem vyhradil 4b. Je tedy možné rozlišit maximálně 16 přístupů do bloku. Logika odsunu bloků je implementována jako stavový automat. Z pohledu jazyka VHDL se však nejedná o stavový automat v pravém slova smyslu. Přechody mezi stavy jsou uskutečňovány asynchroně. Logika cache je implementována pouze v jednom procesu.

Rozhraní entity instrukční cache znázorňuje obrázek 6. Signál ADDR je vstupní adresa z programového čítače. DI je vstupní signál z paměti RAM o šířce 64b, tedy 4 instrukce. Celý blok je načítán v jediném taktu. Na výstupu jsou signály ADDR_OUT a RR pro paměť RAM, samotná instrukce a signál vypadek. Tento signál je přiveden na vstup programového čítače. Je nastaven v případě, že nastane výpadek z IC. V takovém případě je zastaven chod PC.



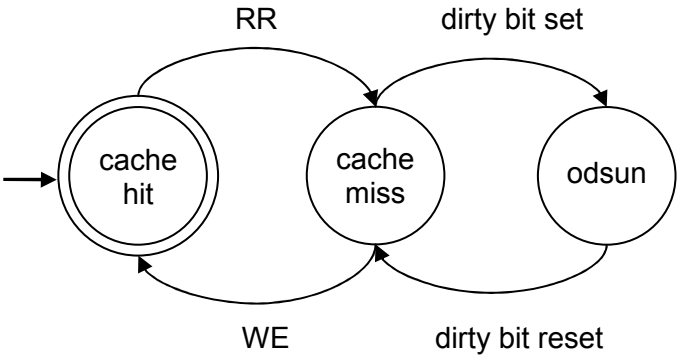
Obr. 6 – Rozhraní entity IC

2.7.2 Datová cache

Datová cache je umístěna na úrovni jednotky ALU. Pokud bychom tedy uvažovali skalární architekturu, jednalo by se o stupeň EX. Tato cache má asynchronní čtení a synchronní zápis (z paměti RAM i pomocí instrukce store). Co se konstrukce týká, zvolil jsem parametry DC shodné s parametry IC. Obsahuje tedy 4 sety po dvou blocích. Každý blok má 4B, celkem 32B. Struktura DC se od IC liší tím, že u každého bloku obsahuje navíc dirty bit. Tento bit indikuje jestli byl daný blok v DC modifikován. Datová cache implementuje tedy navíc algoritmus **write back** pro zpětný zápis modifikovaných dat do RAM. Data jsou tedy zapisovaná do paměti RAM až v okamžiku kdy je potřeba modifikovaný blok odsunout kvůli konfliktu. Datová cache je typu **write allocate** – pokud chci zapisovat data do bloku, který se v cache nenachází, musí být nejprve tento blok načten do DC a teprv pak modifikován.

Vnitřně je DC opět implementována jako asynchronní stavový automat. Oproti IC však obsahuje navíc stav `odsun`. Schéma stavového automatu znázorňuje obrázek 7.

V případě, že jsou načítána nebo zapisována data do bloku, který se nachází v DC, je automat ve stavu cache hit. Pokud jsou data z DC pouze načítána, tak funguje naprosto stejně jako IC. Změna přichází pokud je potřeba data zapsat. Existuje několik případů, které mohou nastat:



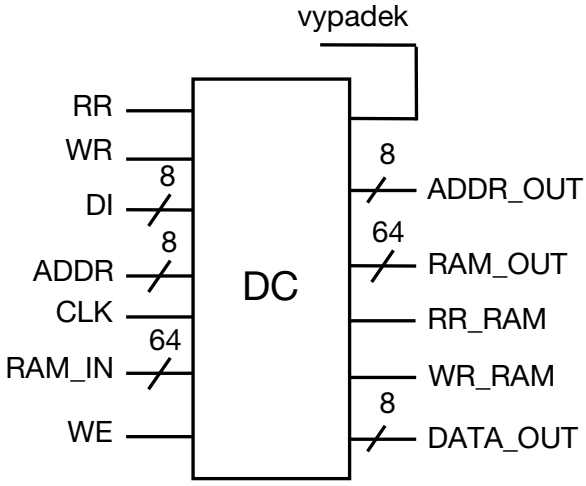
Obr. 7 – Stavový automat datové cache

1) Zápis dat do DC v případě, že DC obsahuje požadovaný blok

V takovém případě, jsou data pouze zapsána a u příslušného bloku je nastaven dirty bit.

2) Zápis dat, DC neobsahuje požadovaný blok

V takovém případě je potřeba požadovaný blok načíst z paměti RAM. Pokud je v datové cache volné místo pro načtení bloku nebo blok na místě zápisu nemá nastaven dirty bit, je z paměti RAM je načten požadovaný blok stejně jak tomu bylo u IC. V tomto případě není potřeba odsouvat blok zpět do RAM. Vnitřní stav DC je změněn zpět na cache hit. Data jsou zapsána a blok je označen dirty bitem. Pokud v datové cache není volné místo pro načtení nového bloku a na místě zápisu nového bloku je blok označený dirty bitem, je nutné odsunout přepisovaný blok zpět do RAM. Vnitřní stav DC je změněn na *odsun*. DC nastaví požadavek na zápis do RAM (signál `WR_RAM`), dále vystaví adresu prvního bytu v odsouvaném bloku a odsouvaná data. S nástupnou hranou CLK jsou data zapsána do paměti RAM. DC zruší nastavení signálu `WR_RAM` a vynuluje dirty bit u odsunutého bloku. Dále změní vnitřní stav na *cache miss* a nastaví požadavek na čtení dat (signál `RR_RAM`) a vystaví adresu načítaného bloku. S nástupnou hranou CLK vystaví paměť RAM požadovaná data a nastaví signál `WE`. S následující nástupnou hranou CLK jsou data zapsána do DC, je zrušen signál `RR_RAM` a vnitřní stav změněn na *cache hit*. Teprv v tuto chvíli je možné zapsat vstupní data z registru do DC. Dirty bit dotčeného bloku je opět nastaven.



Obr. 8 – Rozhraní entity DC

Obrázek 8 znázorňuje rozhraní entity datové cache. Na vstupu oproti IC přibyly signály `RR` a `WR`, kterými instrukční dekodér signalizuje čtení nebo zápis z/do DC. Na výstupu je opět signál vypadek, který je zaveden do programového čítače a v případě nastavení zastavuje jeho činnost. Na výstupu přibyly signály `WR_RAM` a `RAM_OUT` využívané při odsunu bloku zpět do paměti RAM.

Výpadek z datové cache je obsluhován 2 takty pokud nedochází k odsunutí bloku zpět do RAM, 3 takty pokud je odsouván blok zpět do RAM.

2.7.3 Paměť RAM

Paměť RAM je implementována jako pole dlouhé 256 prvků o šířce 16b. Rozhraní RAM je znázorněno na obrázku 9. Jedná se o synchronní komponentu, která obslulhuje výpadky z IC a DC. Data jsou čtena i zapisována synchronně.

Význam jednotlivých signálů:

ADDR_IC…signál z IC - adresa načítaného bytu

RR_IC…signál z IC – požadavek na čtení dat

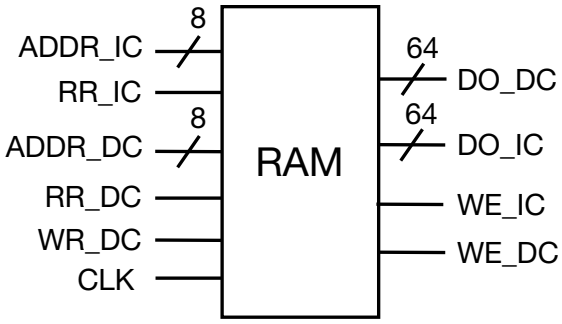
ADDR_DC…signál z DC - adresa načítaného bytu nebo adresa prvního bytu v odsouvaném bloku

RR_DC…signál z DC – požadavek na čtení dat

WR_DC…signál z DC – požadavek na zápis dat

DO_DC, DO_IC…blok dat do datové, instrukční cache.

WE_IC, WE_DC potvrzení o zaslání dat do instrukční a datové cache



Obr. 9 – Rozhraní paměti RAM

Při implementaci paměti RAM jsem se rozhodl provést jedno zjednodušení. Paměť RAM je společná pro instrukce i pro data. Instrukce má šířku 16b, operandy jsou 8mi bitové. Rozhodl jsem se proto implementovat paměť jako pole šířky 16b aby jedné pozici v paměti odpovídala jedna instrukce. Data pro DC jsou v paměti uložena na nižších 8 bitech. Horní polovina rozsahu tak zůstává nevyužitá.

3 VHDL

Jazyk pro popis hardwaru. Zkratka pochází od VHSIC Hardware Description Language. Je standardem IEEE od roku 1987.

Tato kapitola není úplným popisem jazyka VHDL, jedná se spíše o stručný přehled a jsou zde uvedeny pouze ty konstrukce, které byly používány pro programování diplomové práce a se kterými mám osobní zkušenost [1], [2], [3], [4].

Existují tři možnosti, jak činnost hardwaru popsat – behaviorálně, strukturně a dataflow. Tyto způsoby je možné podle potřeby kombinovat.

3.1 Struktura programu

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

Seznam použitých knihoven

```
entity jmeno_entity is
  generic(parametr1 : integer;
          parametr2 : integer := 7);
  port(

  );
end jmeno_entity;
```

Zde je komponenta „popsána“ na úrovni vstupů a výstupů

```
architecture jmeno_architektury of jmeno_entity is

 Místo pro deklaraci signálů, typů, použitých komponent (případě strukturního popisu) atd.

begin

  process(seznam citlivých signálů)
  begin


  end process;
```

Popis chování komponenty (v případě behaviorálního popisu)

Dataflow popis architektury

```
end jmeno_architektury;
```

Knihovny

- `std_logic_1164` definuje mimo jiné datové typy `std_logic` a `std_logic_vector`. Tato knihovna je proto importována do každého zdrojového souboru.
- `std_logic_arith` definuje datové typy `signed`, `unsigned`. Dále definuje základní aritmetické funkce (+, -, *), operátory porovnání (<, <=, >, >=, =, /=), funkce `conv_integer` a `conv_std_logic_vector(typ, počet_bitů)`. Všechny tyto funkce jsou definovány pouze nad výše zmíněnými typy signed a unsigned, dále pak nad typem integer. Knihovna nepracuje s typem `std_logic_vector`. Je to z toho důvodu, že knihovna pracuje se znaménkovými čísly a u typu `std_logic_vector`

není možné rozlišit, jestli se jedná o zápornou nebo kladnou hodnotu. Tuto knihovnu importuji proto pouze v případech, že potřebuji funkci `conv_std_logic_vector`.

- `std_logic_unsigned` definuje stejné aritmetické operace jako knihovna `arith`, stejné relační operátory a dále funkci `conv_integer`. Rozdíl spočívá v tom, že je možné používat operandy typu `std_logic_vector`. S tímto typem je v tomto případě zacházeno jako s typem unsigned.

Rozhraní entity

Klauzule `generic` uvádí generické parametry komponenty, které mohou být inicializovány a použity v kódu jako konstanty. Jiná možnost je inicializovat tyto komponenty při vytváření instance této komponenty, tímto způsobem je tedy možné komponentu parametrizovat. Tato část entity je volitelná.

Klauzule `port` uvádí rozhraní komponenty na úrovni vstupů a výstupů. Pokud komponenta nemá žádné vstupní ani výstupní signály, je tato část vynechána. Signály zde uvedené je možné použít stejným způsobem, jako kdyby byly deklarovány uvnitř architektury s tím rozdílem, že mají definovaný směr vstupů. V programu používám směry in, out a inout.

Architektura

V deklarační části je uvedena deklarace typů, signálů, aliasů, sdílených proměnných, komponent, funkcí, procedur atd.

V těle architektury může být architektura popsána dataflow popisem, jsou zde definovány procesy (v případě behaviorálního nebo smíšeného popisu), vytváří se zde instance a tedy i propojení komponent.

Atributy signálů

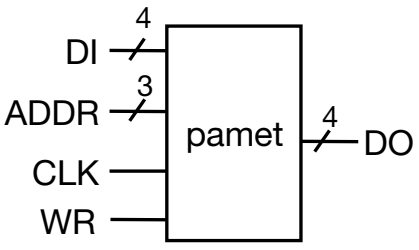
Každý signál má definováno množství atributů, které je možno využít. Atributy použijeme takto: `jmeno_signalu'nazev_atributu`. Nejpoužívanějším atributem je atribut `event`. Vrací hodnotu true, pokud na daném signálu došlo ke změně hodnoty. Dále jsem používal atribut `last_value`. Tento atribut vrací hodnotu signálu před jeho poslední změtnou. Tímto jsem se chtěl vyhnout použití registru. Bohužel tato konstrukce nebyla podporována syntetizátorem.

3.2 Behaviorální popis

Popisuje funkčnost komponenty z hlediska jejího chování.

Př. Behaviorální popis paměti s šířkou slova 4b a velikostí 8 položek.

```
entity pamet is
  port(
     CLK     : in  std_logic;
     WR      : in  std_logic;
     ADDR    : in  std_logic_vector(2 downto 0);
     DI      : in  std_logic_vector(3 downto 0);
     DO      : out std_logic_vector(3 downto 0)
  );
end pamet;
```



Obr. 10Příklad paměti

```
architecture pamet_arch of pamet is
type TPole is array(0 to 7) of std_logic_vector(3 downto 0);
signal pole : TPole;
begin
process(CLK)
begin
if CLK = '1' and CLK'event and WR = '1' then
  pole(conv_integer(ADDR)) <= DI;
end if;
end process;
DO <= pole(conv_integer(ADDR));
end pamet_arch;
```

Rozhraní modelované entity je vidět na obr. 10. Paměť má synchronní zápis a asynchronní čtení.

Proces je spuštěn vždy, když dojde ke změně některého signálu na sensitiviti listu (v tomto případě obsahuje sensitiviti list pouze signál CLK). Sensitiviti list je možné nahradit příkazem `wait on nazev_signalu`. Ekvivalentní zápis k výše uvedenému příkladu by tedy byl:

```
process
begin
wait on CLK;
```

Tato varianta je lepší než použití sensitiviti listu, protože je možné proces pozastavit uprostřed nebo i na několika místech procesu. Bohužel tato konstrukce není podporována syntetizátorem, a proto jsem od používání příkazů typu `wait` upustil.

Příkazy v rámci jednoho procesu se provádí sériově. V jedné architektuře může být několik procesů, které probíhají paralelně a komunikují mezi sebou pomocí signálů nebo sdilených proměnných. Použití sdílených proměnných se mi neosvědčilo, protože komponentu pak nebylo možné syntetizovat.

Řídicí konstrukce a cykly

```
if podmínka then
    …
elsif podmínka then
    …
else
    …
end if;
```

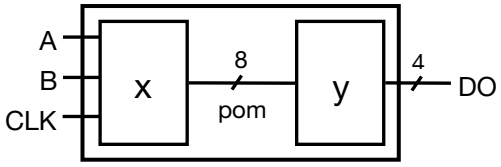
```
case vyraz is
  when hodnota1 =>
    …
  when hodnota2 =>
    …
  when others =>
    …
end case;
```

```
while podmínka loop
    …
end loop;
for i in 0 to 10 loop
    …
end loop;
```

3.3 Strukturální popis

Popisuje komponentu na základě její struktury. Dá se říct, že skládá komponentu z jednodušších částí. Na nejnižší úrovni jsou komponenty ale vždy popsány behaviorálně nebo dataflow.

```
entity komponenta is
  port(
     A   : in  std_logic;
     B   : in  std_logic;
     CLK : in  std_logic;
     DO  : out std_logic_vector(4 downto 0)
  );
end komponenta;
architecture komponenta_arch of komponenta is
signal pom : std_logic_vector(7 downto 0);
begin
X_inst: entity work.x
  port map(A,B,CLK,pom);
Y_inst: entity work.y
  port map(pom,DO);
end komponenta_arch;
```



Obr. 11 – Příklad strukturální entity

Příklad předpokládá, že už máme připravené komponenty x a y, které jsou popsány buď opět strukturálně nebo behaviorálně. Schéma entity je vidět na obr. 11. Ze zdrojového kódu je vidět způsob vytváření instancí komponent a způsob mapování signálů. Existují dvě možnosti vytváření instancí komponent:

Přímá instance komponent

Tento postup je použit ve zdrojovém kódu výše. Uvedu název instancované entity spolu s knihovnou, ve které se entita nachází. Uživatelské komponenty jsou standardně umisťovány do knihovny `work`. Tento postup je nejjednodušší a používám ho všude ve svých zdrojových kódech.

Instance pomocí komponent

V deklarační části architektury se uvede rozhraní instancovaných komponent. Toto rozhraní je identické s rozhraním instancované entity s tím rozdílem, že je místo klíčového slova `entity` použito klíčové slovo `component`. Tento způsob je demonstrován na následujícím zdrojovém kódu. Používám opět architekturu z obr. 11.

```
architecture komponenta_arch of komponenta is
component x is
  port(
     A   : in  std_logic;
     B   : in  std_logic;
     CLK : in  std_logic;
     DO  : out std_logic_vector(7 downto 0)
  );
end component x;
component y is
  port(
     DI : in  std_logic_vector(7 downto 0);
     DO : out std_logic_vector(3 downto 0)
  );
end component y;
signal pom : std_logic_vector(7 downto 0);

begin

X_inst: x
  port map(A => A,B => B,CLK => CLK,DO => pom);

Y_inst: y
  port map(DI => pom,DO => DO);

end komponenta_arch;
```

Ve zdrojovém kódu je vidět způsob deklarace komponenty i způsob vytvoření samotné instance – uvádí se název instance (červeně) a název importované komponenty. Tento způsob má tu výhodu, že je možné instancovat komponenty, které ještě vůbec neexistují. Například pokud je rozsáhlá architektura rozdělena mezi více vývojářů a ještě nejsou naprogramovány všechny komponenty. Při překladu nevznikne chyba, pouze signály chybějících komponent budou mít nedefinovanou hodnotu.

Propojení instancovaných komponent je opět možné provést dvěma způsoby

Zkrácená definice

Tento způsob je uveden na prvním příkladu. V klauzuli `port map(…)` jsou po řadě uvedeny signály hlavní komponenty tak, jak mají být namapovány na signály instancované komponenty. Signály hlavní komponenty musí být uvedeny v tom pořadí, v jakém jsou uvedeny v rozhraní instancované komponenty. Dá se tedy říct, že signály hlavní komponenty jsou mapovány na signály instancovaných komponent.

Úplná definice

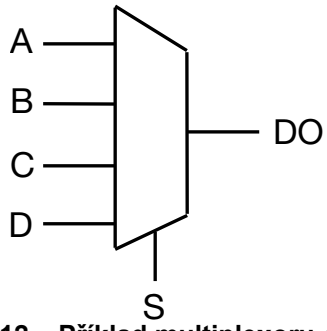
Tento způsob je ukázán na druhém příkladě. Používá opačný postup než zkrácená definice. Signály instancovaných komponent jsou mapovány na signály hlavní komponenty. Na pořadí mapování v klauzuli `port map`(…) v tomto případě nezáleží.

3.4 Dataflow popis

Tímto stylem popisu je možné popsat komponentu paralelně. Výstupy jsou přepočítávány v každém simulačním kroku. Tuto variantu používám při popisu asynchronních komponent,

např. ALU. Příkazy `case` a `if` jsou u tohoto způsobu popisu odlišné od těch, které se používají v behaviorálním popisu. Následuje příklad dataflow popisu multiplexoru 4-1, který můžeme vidět na obrázku 12.

```
entity MX4_1 is
  port(
     A  : in  std_logic;
     B  : in  std_logic;
     C  : in  std_logic;
     D  : in  std_logic;
     S  : in  std_logic_vector(1 downto 0);
     DO : out std_logic
  );
end entity MX4_1;

architecture MX4_1_arch of MX4_1 is
with S
  DO <= A when "00";
        B when "01";
        C when "10";
        D when "11";
        X when others;
end MX4_1_arch;
```



Obr. 12 – Příklad multiplexoru 4-1

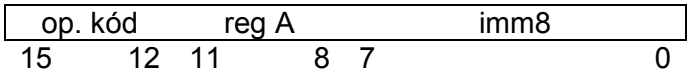
4 Instrukční sada

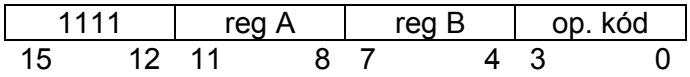
Tato instrukční sada je inspirována [1] a je implementována všemi verzemi procesoru. Jedná se o **load/store** instrukční sadu – přístup do pamětí je možný pouze pomocí instrukcí LD/ST. Instrukce má pevnou délku – 16b. Architektura umožňuje dva adresovací režimy – adresace pomocí registrů a přímý operand. Tyto dva režimy mají význam pouze u aritmeticko-logických instrukcí. Ostatní skupiny využívají vždy pouze jednu variantu. Přepínání mezi těmito adresovacími režimy je řešeno pomocí speciálního operačního kódu.

Operační kód je v zásadě pouze na 4b. Implementovaných instrukcí je ale ve skutečnosti víc než 16, proto jsou rozděleny do několika skupin. Operační kód (4b) je pak přiřazen celé skupině a rozlišení instrukcí probíhá na základě druhé části op. kódu umístěného na volné pozici v instrukci.

Pro adresaci registrů jsou vyhrazeny 4b, což předurčuje velikost registrového pole na 16 položek. Operandy mají velikost 8b.

1. skupina – aritmeticko-logické instrukce, instrukce CMP, LD a ST



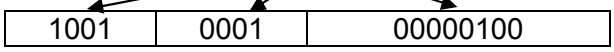


Pro zkrácení délky instrukce je adresa zdrojového operandu shodná s adresou cílového operandu. Výše uvedené tabulky zobrazují umístění operačního kódu, adresy zdrojových operandů a okamžitý operand. Pokud je přítomný okamžitý operand, je operační kód na horních 4 bitech instrukce. Pokud jsou oba operandy v registrech, je na místě operačního kódu hodnota 1111b a samotný operační kód je umístěn v dolních 4 bitech. V obou případech jsou operační kódy pro shodné instrukce stejné.

Pokud se jedná o instrukce LD a ST, obsahuje registr A vstupní data do datové cache, okamžitý operand nebo operand z registru B pak obsahuje adresu do datové cache. Uvedená skutečnost je trochu matoucí při převodu zdrojového kódu v asembleru (konkrétně isntrukcí LD a ST) do binárního kódu.

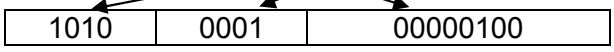
Př.

asm: LD rb,04h



Instrukce uloží do registru rb data z DC z adresy 04h. První operand je cílový, druhý operand je zdrojový (tak jak je v asembleru zvykem).

asm: ST 04h,rb



V případě instrukce store je cílový operand adresa do instrukční cache a musí být uveden na pozicích 7…0 (v případě okamžitého operandu) nebo 7…4 (v případě, že je adresa do DC uložena v registru).

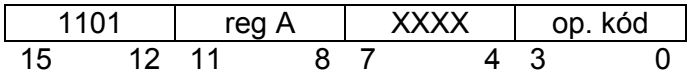
Stejné pravidlo platí i pro instrukce IN a OUT.

Asm	Binárně
ADD	0000
ADC	0001
SUB	0010
SBC	0011
AND	0100
OR	0101
XOR	0110
MOV	0111
CMP	1000
LD	1001
ST	1010
IN	1011
OUT	1100

Tabulka 2 - Zakódovaní aritmeticko-logických instrukcí

Instrukce CMP je implementována stejně jako instrukce SUB (provádí se odčítání operand a – operand b). Výsledek není zapsán do registrového pole, pouze jsou nastaveny příznaky carry a zero.

2. skupina – instrukce posuvů a rotací

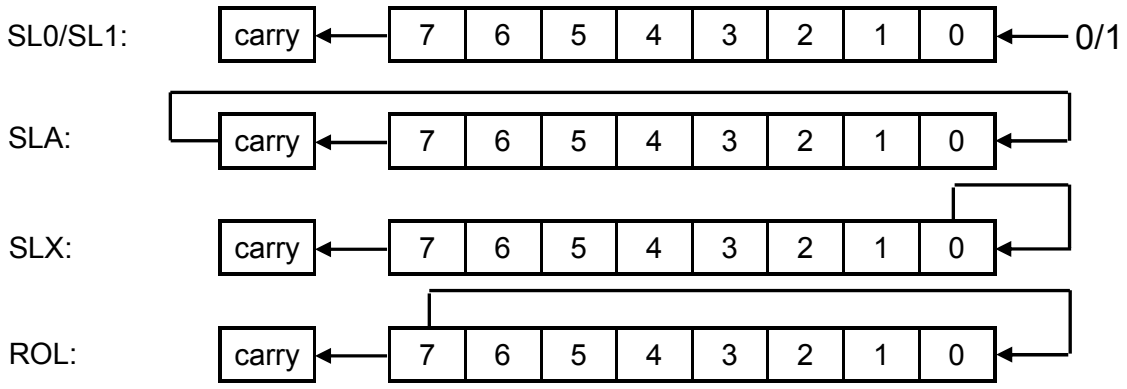


Operační kód 1101b označuje skupinu instrukcí s posuvy a rotacemi. Operační kód jednotlivých instrukcí je pak opět v nejnižších 4 bitech. Bity na pozicích 7…4 mohou mít libovolnou hodnotu. Ve zdrojových kódech jsou vždy nastaveny na hodnotu 0000b. Mezi tyto instrukce jsem zařadil i instrukci NOOP. Instrukce je implementována pouze v procesorech s pamětí RAM. Její význam bude vysvětlen v kapitole 5.3.

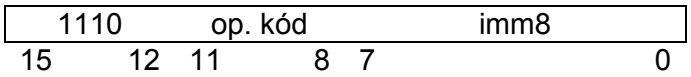
Asm	Binárně
SL0	0000
SL1	0001
SLA	0010
SLX	0011
ROL	0100
SR0	0101
SR1	0110
SRA	0111
SRX	1000
ROR	1001
NOOP	1111

Tabulka 3 – Zakódování instrukcí pro posuvy a rotace

Instrukce pro posuvy mají tento význam:



3. skupina – skoky



Instrukční architektura je navržena tak, že podporuje pouze absolutní skoky (obsah programového čítače je nahrazen cílovou adresou skoku). Cílová adresa skoku je uvedena v okamžitém operandu. Operační kód pro tuto skupinu je 1110b. Operační kód pro rozlišení jednotlivých instrukcí je umístěn na pozici 11…8.

Asm	Binárně
JMP	0000
JZ	0001
JNZ	0010
JC	0011
JNC	0100
CALL	0101
RET	0110

Tabulka 4 – Zakódování skokových instrukcí

Instrukce RET (návrat z procedury) může mít na pozici okamžitého operandu libovolnou hodnotu. Ve zdrojových kódech je vždy nastavena na 00h.

5 Architektury procesorů

5.1 Subskalární architektura

Subskalární architektura byla první PC architekturou. Neobsahuje žádný paralelismus, provádění instrukcí probíhalo v jediném taktu. Zpracování následující instrukce mohlo být započato teprve v momentě, když byla dokončena současná instrukce. Časový diagram takovéto architektury je znázorněn v tabulce 5. V jediném taktu probíhalo načtení, dekódování, zpracování a zápis výsledku. Některé instrukce nepotřebují projít všemi stupni linky, ale zpracování následující instrukce stejně nemůže být započato dřív než v následujícím taktu. I subskalární architektury se samozřejmě lišily. Jiný způsob řízení chodu procesoru bylo pomocí stavového automatu, kdy se aktivovala pouze ta část procesoru, kterou aktuálně daná instrukce využívala. V tomto případě trvalo zpracování instrukcí několik taktů, ale frekvence mohla být vyšší, takže doba zpracování jedné instrukce byla stejná jako u první varianty.

Takt	1				2				3				4			
I1																
I2																
I3																
I4																

Tabulka 5 – Subskalární zpracování instrukcí

Pro pozdější srovnání se skalárním procesorem budeme počítat, že procesor stráví v každém stupni architektury ¼ taktu. Pokud bychom tedy subskalární architekturu rozdělili na jednotlivé stupně, mohla by frekvence procesoru vzrůst teoreticky 4x a mohli bychom počítat, že procesor bude potřebovat 4 takty pro provedení jedné instrukce.

Průměrné CPI = 4

5.2 Návrh subskalární architektury bez cache

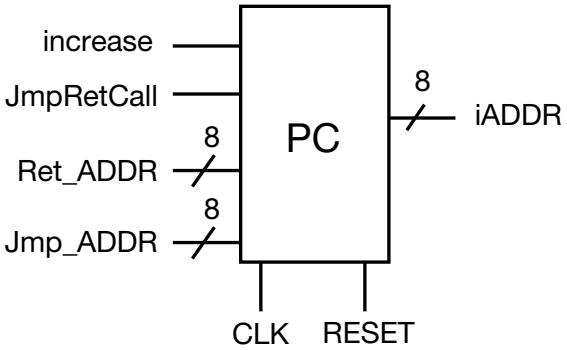
Subskalární architektura je implementována pro referenci k pozdější skalární architektuře. Její návrh vychází z [1]. Návrh jsem mírně upravil a implementoval tak, aby výsledný kód bylo možné syntetizovat. Procesor nemá logiku instrukční a datové cache. Tyto cache jsou v této verzi procesoru implementovány pouze jako pole.

Jedná se o 8b architekturu typu load/store. Přístup do paměti mají pouze instrukce LD a ST. Celý procesor je řízen pouze dekodérem. V každém taktu je provedena jedna instrukce. Vyjímku tvoří pouze instrukce zapisující data do synchronních pamětí, jako je registrové pole, zásobník atd. V takovém případě je zápis proveden s nástupnou hranou dalšího taktu. Procesor implementuje instrukční sadu definovanou v kapitole 4.

5.2.1 Popis částí procesoru

Programový čítač

Programový čítač (PC), jehož rozhraní ukazuje obr. 13, plní funkci ukazatele do instrukční cache. Signál RESET nuluje PC, pokud není RESET nastaven, tak je s každou nástupnou hranou vystavena korektní adresa instrukce, která má být



Obr. 13 – Instrukční cache

zpracována v dalším taktu. Toto platí i pro instrukce, které následují po skokových instrukcích.

Význam signálů

`increase, JmpRetCall` - Signály z dekodéru informují PC, jestli nastala změna toku instrukcí (v případě skoků) nebo jestli je tok instrukcí normální.

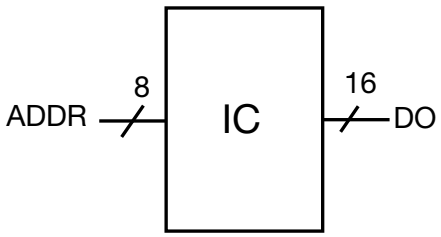
`Ret_ADDR` – sinál ze zásobníku návratových adres, tento signál je použit jako návratová adresa při zpracování instrukce `ret` (návrat z procedury).

`Jmp_ADDR` – signál z dekodéru určující cílovou adresu skoku nebo volání procedury

`iADDR` – výstupní signál adresy do instrukční cache

Instrukční cache

V této verzi procesoru je implementována jako paměť typu ROM s 256 položkami šířky 16b. Paměť má asynchronní čtení. Rozhraní entity můžeme vidět na obr. 14. Na vstupu dostává adresu instrukce, na výstupu je instrukce pro dekodér. IC je ve vzorovém kódu níže inicializována na samé nuly, ve zdrojových kódech odevzdávaných na CD obsahuje instrukce algoritmu bubble sort.

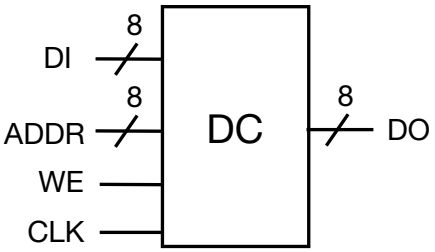


Obr. 14 – Instrukční cache

```
entity IC is
  port(
     ADDR : in  std_logic_vector(7 downto 0);
     DO   : out std_logic_vector(15 downto 0)
  );
end entity IC;
architecture IC_arch of IC is
type TPole is array (0 to 255) of std_logic_vector(15 downto 0);
signal pole : TPole := (others => X"0000");
begin
  DO <= pole(conv_integer(addr));
end IC_arch;
```

Datová cache

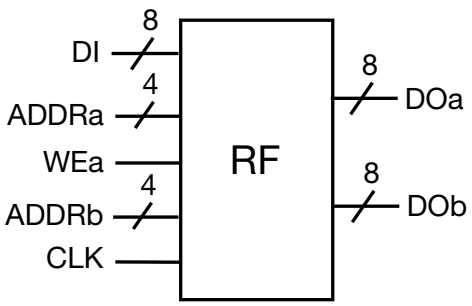
V této verzi implementovaná jako paměť pro čtení a zápis s 64 položkami šířky 8b (stejně jako IC je DC pouze pole signálů typu std_logic_vector). Paměť má synchronní zápis a asynchronní čtení. Signál `WE` povoluje zápis do datové cache. Ve zdrojových kódech DC obsahuje deset čísel, které jsou řazeny pomocí algoritmu bubble sort. Zbylá část DC je inicializována na samé nuly. Rozhraní entity je znázorněno na obr. 15.



Obr. 15 – Datová cache

Pole registrů

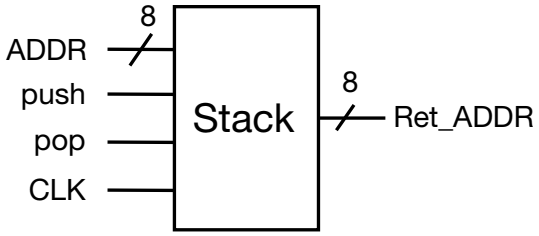
Je koncipováno jako dvouportová paměť s 16ti položkami o šířce 8b. Signál `ADDRa` slouží adresa pro načtení prvního operandu a zároveň jako adresa pro zápis výsledku zpět do registrového pole. Signál `WEa` povoluje zápis do RF. Signál `ADDRb` udává adresu druhého zdrojového operandu. Paměť má synchronní zápis a asynchronní čtení. Rozhraní entity je znázorněno na obr. 16.



Obr. 16 – Registrové pole

Zásobník návratových adres

Slouží pro uložení adresy instrukce volání procedury - `call`. Zápis a odstranění adresy ze zásobníku je řízeno z dekodéru řídícími signály `push` a `pop`. Zápis i „odstranění“ adresy ze zásobníku probíhá synchronně, na výstupu zásobníku je ale vždy asynchronně vystavena poslední návratová adresa. V případě, že je zásobník prázdný, tak je vystavena hodnota v zásobníku z adresy 00h. Rozhraní entity znázorňuje obr. 17.



Obr. 17 – Zásobník návratových adres

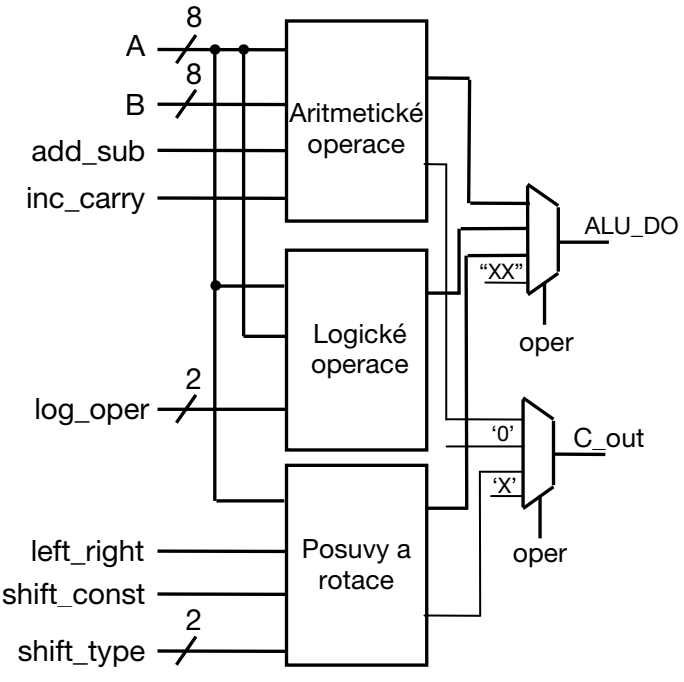
```
architecture stack_arch of stack is
type TPole is array(0 to 255) of std_logic_vector(7 downto 0);
signal zasobnik : TPole := (X"01", others => X"00");
begin

process(CLK,zasobnik)
variable ukazatel : std_logic_vector(7 downto 0) := X"FF";
begin

if CLK = '1' and CLK'event then
  if push = '1' then
    zasobnik(conv_integer(ukazatel)) <= ADDR_IN;
    ukazatel := ukazatel - '1';
  elsif pop = '1' then
    ukazatel := ukazatel + '1';
  end if;
end if;

ADDR_OUT <= zasobnik( conv_integer(ukazatel + '1') );

end process;
```

Zásobník je implementován jako pole o velikosti 255 a má záměrně první položku inicializovánu na odlišnou hodnotu od ostatních aby bylo ze simulace zřejmé, že při prázdném zásobníku je na výstupu položka z jeho dna. Ukazatel zásobníku ukazuje vždy na první volnou položku.

Aritmeticko-logická jednotka ALU

Plní základní celočíselné operace – aritmetické, logické, instrukci MOV, posuvy a rotace.

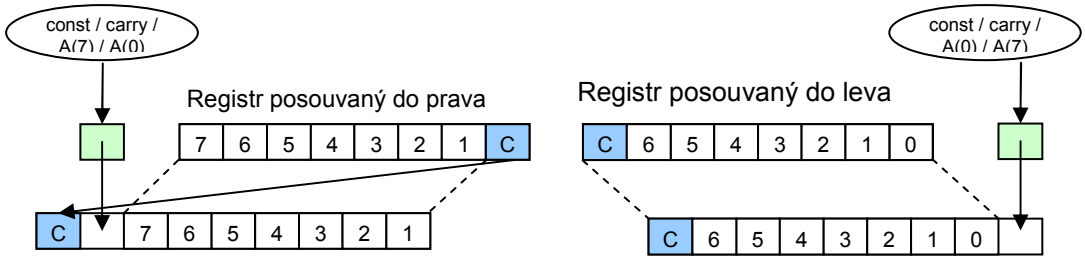
ALU je implementována dataflow popisem. Rozhraní entity a způsob organizace ukazuje obrázek 18. Způsob implementace posuvů a rotací je znázorněn na obrázku 19.



Obr. 18 - Jednotka ALU

Význam signálů:

`add_sub` – přičítání/odčítání
`inc_carry` – včetně carry/bez carry
`log_oper` – vybírá jednu ze 4 logických operací (mezi logické operace je zahrnuta i operace mov)
`left_right` – posuv nebo rotace bude doleva/doprava
`shift_sel` – výběr typu posuvu rotace
`shift_const` – nasouvaná konstanta
`oper` – přepíná na výstup jednu z operací



Obr. 19 – implementace posuvů a rotací dataflow popisem

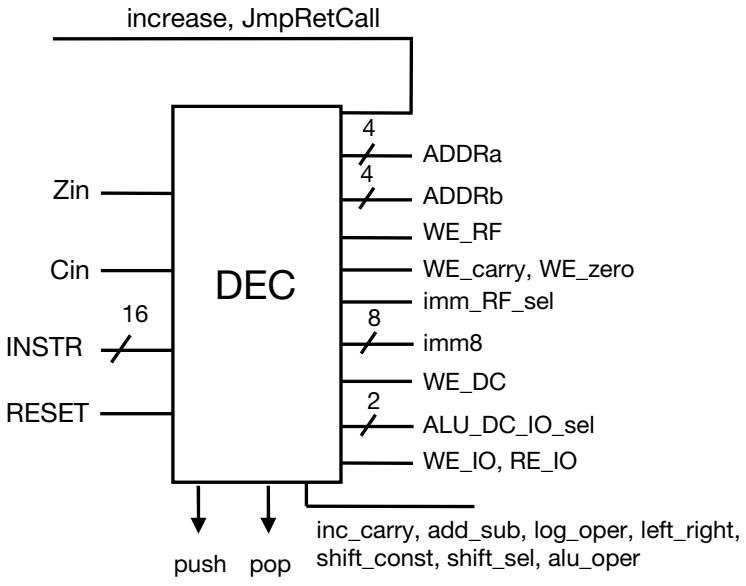
Aby byl možný dataflow popis i u jednotky pro posuvy a rotace, vytvořil jsem si dva pomocné signály: MSB pro posuvy doprava, LSB pro posuvy doleva. Na obrázku 19 jsou znázorněny zeleně. Signál MSB mi reprezentuje nejvíce významový bit registru, který posouvám doprava. V závislosti na typu posuvu (ten je určen signálem `shift_type`) je do tohoto signálu přiřazena konstanta, registr carry, zůstane v něm nejvíce významový bit nebo je do něj přiřazen nejméně významový bit. Obdobně je přiřazena hodnota signálu LSB v případě posuvů doleva. Výstup je přiřazen do 9b registru, kde na nejvíce významovém bitu je vždy hodnota carry bitu a na ostatních bitech je posunutý registr. Z tohoto registru je pak už snadno vytvořen výstup z jednotky ALU.

Instrukce MOV je implementována jako součást logických operací. Pro její provedení totiž stačí pouze poslat na výstup druhý operand.

Instrukční dekodér

Je klíčová komponenta celého modelu počítače. Dekóduje instrukci načtenou z instrukční cache na základě operačního kódu a nastavuje veškeré potřebné řídicí signály. Ve VHDL je implementován jako jeden veliký case, který se větví podle operačního kódu instrukce. Pokud je nastaven signál RESET, zasílá dekodér defaultní řídicí signály.

Na vstup dostává 16b instrukci načtenou z IC, 1b vstupy carry a zero, které jsou využívány pro podmíněné skoky. Rozhraní entity je vidět na obrázku 20.



Obr. 20 – Instrukční dekodér

Význam signálů:

`increase, JmpRetCall` - signály pro programový čítač, udávají, jestli je aktuálně zpracovávaná instrukce skoková nebo jiná.

`ADDRa,ADDRb` – adresy operandů do registrového pole, `ADDRa` je zároveň adresou cílového operandu, pokud je výsledek zapisován zpět do RF

`WE_RF` – povoluje zápis do registrového pole

`WE_carry, WE_zero` – povolují zápis do registrů carry a zero

`imm_RF_sel` – signál pro multiplexor, který přepíná mezi druhým operandem z registrového pole a mezi okamžitým operandem reprezentovaným signálem `imm8`

`WE_DC` – povoluje zápis do datové cache

`ALU_DC_IO_sel` – signál pro multiplexor na výstupu, který přepíná výstupy z jednotek ALU, DC nebo z externího zařízení

`WE_IO, RE_IO` – povolují zápis/čtení do/z externího zařízení

`push, pop` – signály pro zásobník návratových adres. Přikazují zásobníku uložit adresu instrukce na vstupu a dekrementovat ukazatel zásobníku, resp. vystavit adresu ze zásobníku na výstup a inkrementovat ukazatel.

Standardní nastavení signálů

Výstupní signály dekodéru je potřeba inicializovat na určitou hodnotu. Není možné např. nechat signál, povolující zápis, nastavený na hodnotu 1 protože by došlo s první nástupnou hranou k zápisu nesmyslného operandu do paměti. Řídicí signály jsou standardně nastaveny takto:

```
imm_RF_sel <= '1';              --operand 2 do ALU bude imm8
WE_DC <= '0';                   --zákaz zápisu do DC
ALU_DC_IO_sel <= "00";          --je na výstupu výsledek z ALU
WE_RF <= '0';                   --zakázaný zápis výsledku do RF
WE_carry <= '0';                --zakázaný zápis výsledku do carry
WE_zero <= '0';                 --zakázaný zápis výsledku do zero
RE_IO <= '0';                   --zakázané čtení z IO portu
WE_IO <= '0';                   --zakázaný zápis na IO port
oper <= "00";                   --aritmetika
add_sub <= '1';                 --sčítání
inc_carry <= '0';               --bez carry
log_oper <= "00";               --logická operace AND
left_right <= '1';              --posuv do leva
shift_type <= "00";             --nasouvá se konstanta
shift_const <= '0';             --nasouvaná konstanta je nula
increase <= '1';                --přičítá se jednička
JmpRetCall <= '0';              --PC = PC + 1
push <= '0';                    --do zásobníku se nebude nic zapisovat
pop <= '0';                     --ze zásobniku se nebude nic číst
```

Červené signály musí být nastaveny tak, jak je uvedeno. Signály povolující zápis musí mít na začátku hodnotu nula, jinak by došlo s první nástupnou hranou k zápisu náhodných hodnot do pamětí a registrů. Signály `push` a `pop` opět musí mít oba nastavenou hodnotu 0, aby nedošlo s první nástupnou hranou ke špatné funkci zásobníku. Uvedená kombinace signálů `increase` a `JmpRetCall` znamená normální čítání instrukcí (tedy čítání neskokových instrukcí). Signály `WE_IO` a `RE_IO` musí být opět na hodnotě 0, aby nedošlo ke špatné komunikaci s externím zařízením.

Zelené signály mohou mít libovolnou hodnotu. Standardně jsou nastavené tak, aby byla provedena instrukce ADD registru a okamžitého operandu.

Registry carry a zero

Jsou implementovány jako registry se synchroním zápisem, pokud je nastaven příslušný povolovací signál.

Ostatní komponenty

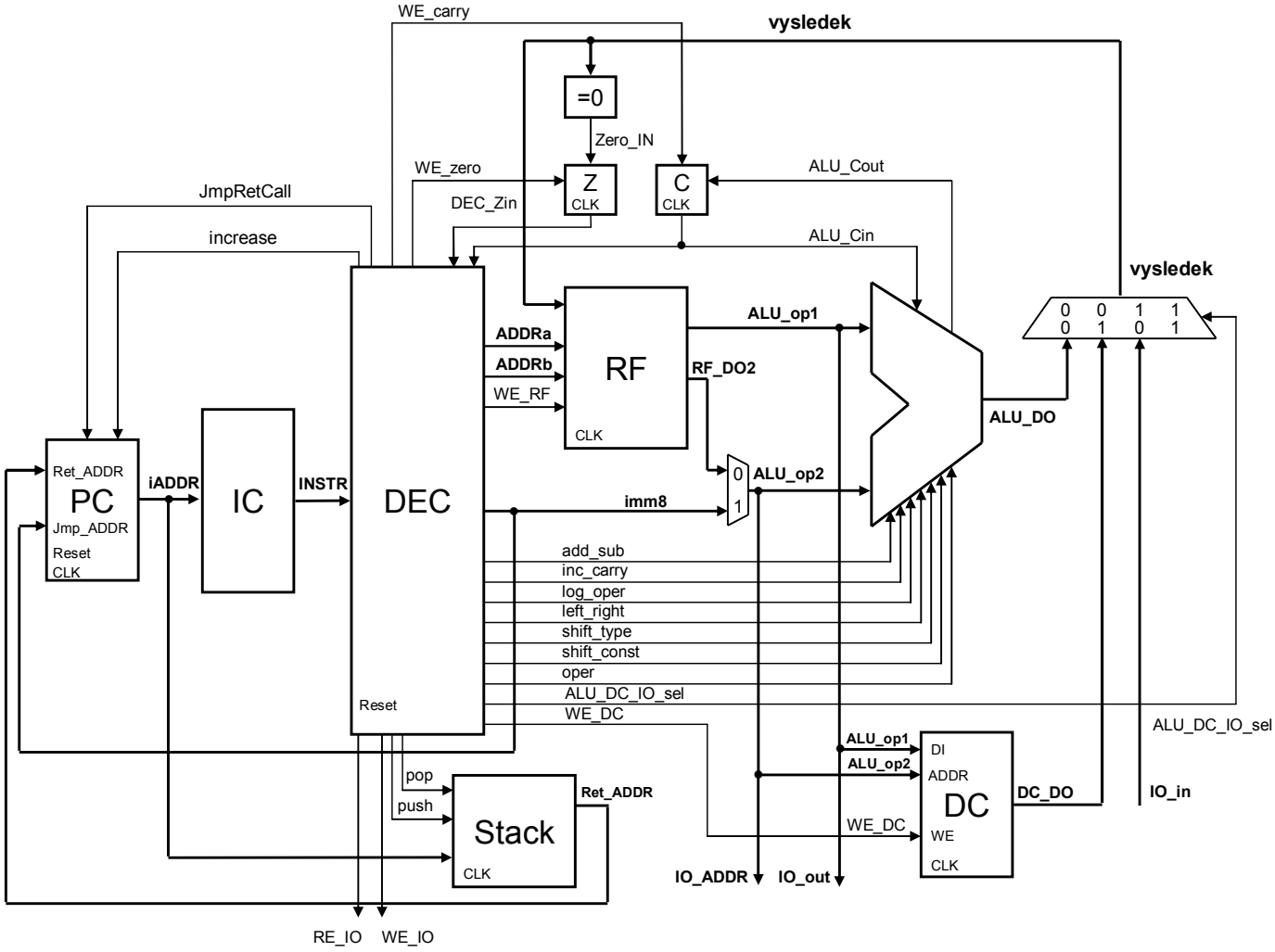
Komparátor na nulu – porovnává výsledek s nulou, v případě shody nastaví signál na výstupu na hodnotu 1. Výstup komparátoru je připojen na vstup registru zero.

MX2_1 – multiplexor přepínající mezi okamžitým operandem (`imm8`) a druhým operandem z registrového pole (signál `rf_do2`). Na vstupu dostává řídicí signál z dekodéru (`imm_RF_sel`) a vstupní hodnoty.

MX4_1 – multiplexor přepínající mezi výstupy z ALU, DC, a externího vstupu. Řídicí signál ALU_DC_IO_sel přichází z dekodéru.

Celý procesor se tedy skládá z 12 komponent. Dále je potřeba komponenta, která jednotlivé komponenty propojí a test bench pro hlavní komponentu. Celkem se tedy model procesoru skládá ze 14 zdrojových souborů.

5.2.2 Schéma procesoru a popis funkčnosti



Obr. 21 – Architektura skalárního procesoru

Obrázek 21 zobrazuje architekturu subskalárního procesoru přesně tak, jak je implementována. Názvy propojovacích signálů odpovídají skutečným názvům tak, jak jsou definovány v souboru „SubSkalar.vhd“.

Význam jednotlivých signálů:

Signál	Význam
iADDR	adresa do instrukční cache 8b
INSTR	načtená instrukce z IC 16b
increase	0 - adresa PC se nebude inkrementovat 1 - adresa PC se bude inkrementovat o 1
JmpRetCall	0 - aktuální instrukce není skoková 1 - aktuální instrukce je skoková
add_sub	0 - odčítání 1 - sčítání
inc_carry	0 - bez carry 1 - včetně carry
log_oper	00 - and 01 - or 10 - xor 11 - mov
left_right	0 - směr doprava 1 - směr doleva
shift_type	00 - nasouvat se bude konstanta (signál shift const) 01 - nasouvat se bude carry 10 - nasouvat se bude LSB/MSB (v případě směru doleva/doprava) 11 - nasouvat se bude MSB/LSB (v případě směru doleva/doprava)
shift_const	nasouvaná konstanta
oper	00 - aritmetika 01 - logika 10 - posuvy a rotace 11 - zakázaná kombinace
ALU_DC_IO_sel	00 - výstup z ALU 01 - výstup z DC 10 - vstup z externího zařízení 11 - zakázaná kombinace
WE_RF, WE_carry, WE_zero, WE_DC	0 – zápis zakázán 1 – zápis povolen
RE_IO, WE_IO	povolují/zakazují čtení z/do externího zařízení

Tabulka 6 – význam signálů u subskalární architektury

Stručný popis funkčnosti

Na počátku je nastaven signál RESET. Tím je vynulován programový čítač a na jeho výstupu se objeví adresa 00h. Ta je asynchronně zaslána na vstup IC, ze které je asynchronně načtena instrukce a zaslána dekodéru. Dekodér vystaví adresy zdrojových operandů a hodnotu okamžitého operandu. V závislosti na operačním kódu nastaví všechny řídící signály. Pokud se jedná o skokovou instrukci, tak tímto „užitečná činnost“ procesoru končí. Pokud se jedná o aritmeticko-logickou instrukci, je provedena příslušná operace, výsledek je vystaven na vystup ALU, případně je nastaven výstup pro carry. Pokud se jedná o instrukci load, je vystavena adresa a na výstupu se objeví načtený operand z DC. V případě instrukce ST, je vystavena adresa i data. Zápis dat do RF a DC proběhne s následující nástupnou hranou CLK.

5.2.3 Příklady

5.2.3.1 Bubble sort

Pro demonstraci funkčnosti procesoru jsem napsal algoritmus bubble sort, řadící prvních deset položek datové cache.

Datová cache je inicializovaná na tyto hodnoty:

Adresa hex	00	01	02	03	04	05	06	07	08	09	0A	0B
Obsah hex	7	3	1	5	0A	0F	2	10	3	4	0	…
Obsah dec	7	3	1	5	10	15	2	16	3	4	0	…

Zdrojový kód

Adresa hex	Instrukce hexadecimálně	Význam instrukce	
0	7200	init:	mov rc,0
1	7301		mov rd,1
2	F029		ld ra,rc
3	F139		ld rb,ra
4	F018	zac:	cmp ra,rb
5	E309		jc pokr
6	F12A		st rc,rb
7	F03A		st rd,ra
8	F029		ld ra,rc
9	0301	pokr:	add rd,1
0A	830A		cmp rd,10
0B	E10E		jz konec1
0C	F139		ld rb,rd
0D	E004		jmp zac
0E	0201	konec1:	add rc,1
0F	F327		mov rd,rc
10	0301		add rd,1
11	8209		cmp rc,9
12	E116		jz konec2
13	F029		ld ra,rc
14	F139		ld rb,rd
15	E004		jmp zac
16	E000	konec2:	jmp init

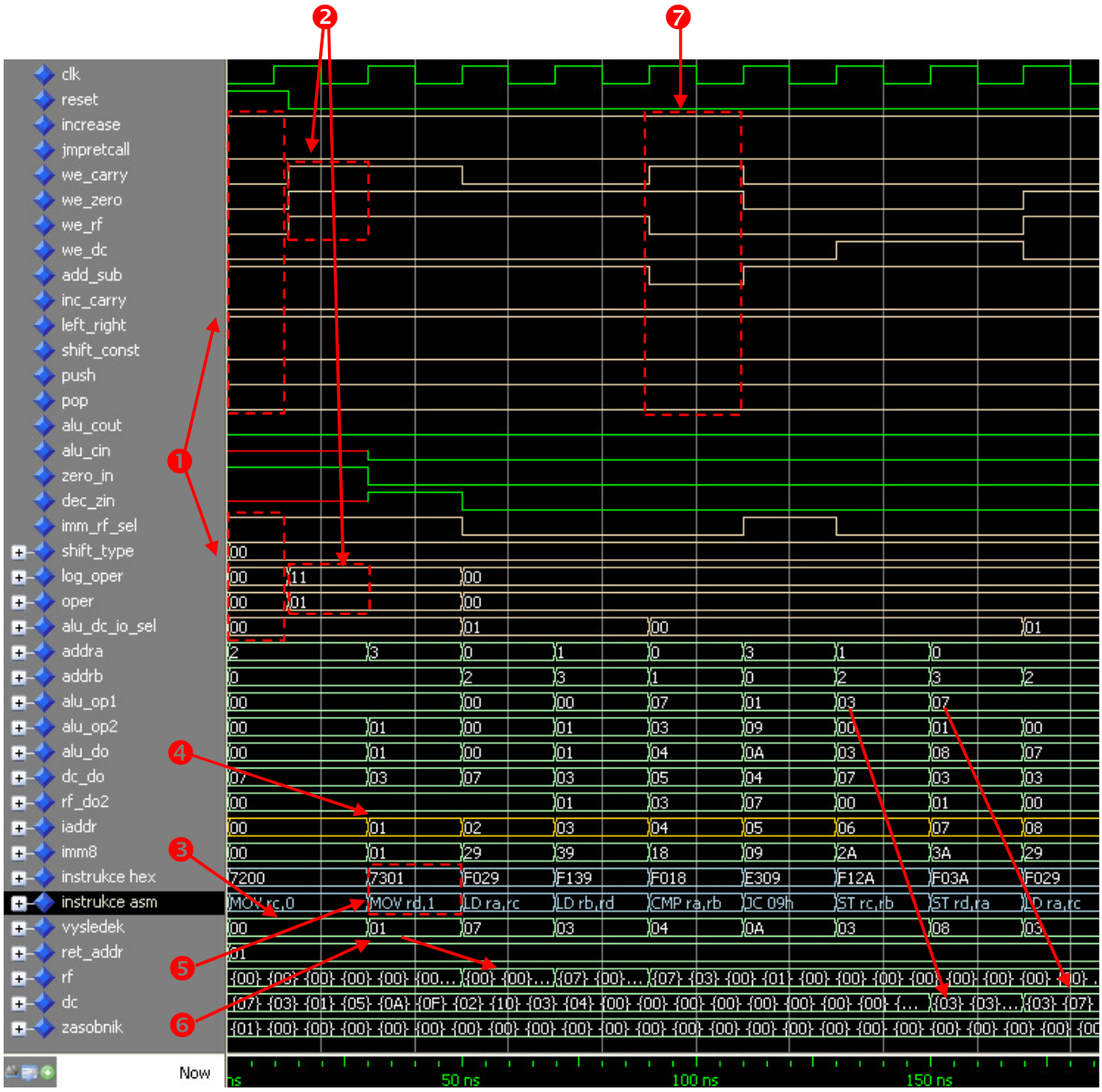
Vnitřní for cyklus

Vnější for cyklus

Registr rc slouží jako index ve vnějším cyklu, registr rd jako index ve vnitřním cyklu. Do registru ra se načítá prvek z vnějšího cyklu, do registru rb načítám prvky ve vnitřním cyklu.

Wave diagram

Na obrázku 23 je znázorněn wave diagram provádění několika prvních instrukcí algoritmu Bubble sort. Řídicí signály dekodéru jsou zobrazeny růžově, adresa instrukce tmavožlutě a instrukce (ve tvaru hex i asm) jsou zobrazeny modře. Barvy ostatních signálů zůstaly nezměněny.



Obr. 23 Wave diagram algoritmu Bubble sort

Na počátku simulace je nastaven signál RESET. To způsobí vynulování programového čítače (signál iADDR = 00). Dekodér v tuto chvíli posílá na výstup standardní řídicí signály (❶). IC má už v okamžiku nastavení RESETu na vstupu adresu 00h (signál iADDR), a proto asynchroně vystaví na výstup první instrukci. V tomto případě se jedná o instrukci MOV rc,0 (7200h). Dekodér vystaví adresy obou operandu a hodnotu okamžitého operandu (bity 11..8 = ADDRa, bity 7..4 = ADDRb, bity 7..0 = imm8). Z registrového pole jsou asynchroně načteny oba operandy (signály ALU_op1 a RF_DO2). Multiplexor přepínající mezi okamžitým operandem a druhým operandem z RF je standardně nastaven, aby propouštěl okamžitý operand (signál imm_RF_sel = 1). Druhý operand pro jednotku ALU (signál ALU_op2) bude tedy signál imm8.

V okamžiku vynulování RESETu vystaví dekodér korektní řídicí signály pro aktuálně zpracovávanou instrukci. Jsou tedy povoleny zápisy do registrového pole, registrů carry a zero (signály WE_RF, WE_carry, WE_zero), dále je nastaven signál pro výběr logické operace (signál log_oper) a signál pro výběr logické jednotky jako výstupu ALU (signál oper) (❷). Pozn. instrukce MOV je implementována jako logická operace.

Protože je téměř celý procesor implementován asynchronně, tak je ještě ve stejném taktu vystaven výsledek (❸).

S další nástupnou hranou hodinového signálu je zapsán výsledek předchozí instrukce do registrového pole (toto z diagramu není patrné, protože výsledek předchozí instrukce byl 0 a registrové pole je inicializováno na samé nuly). Dále jsou pak zapsány hodnoty do registrů carry a zero. Toto je patrné z výstupů těchto registrů (signály alu_cin a dec_zin).

S nástupnou hranou je také inkrementován PC (❹). Opět je asynchronně vystavena instrukce (7301h = MOV rd,1) (❺), dekodérem dekódována (řídicí signály zůstávají v tomto případě shodné, protože se jedná opět o instrukci MOV), na výstup je poslán výsledek, který je s následující nástupnou hranou zapsán do RF (❻). Zajímavá je instrukce CMP, která je implementována jako instrukce SUB s tím rozdílem, že má zakázaný zápis výsledku do RF (❼). Na konci diagramu můžeme vidět prohození prvních dvou prvků řazené posloupnosti.

Výstup z Transcriptu

```
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/dc
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an
arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/dc
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/rf
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an
arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/rf
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/rf
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an
arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/rf
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/ic
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an
arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /subskalar_tb/uut/ic
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
#    Time: 0 ns  Iteration: 2  Instance: /subskalar_tb/uut/alu
```

Analýza varování z hlediska simulačních kroků

Varování týkající se nultého simulačního kroku je způsobeno tím, že na vstupu v tomto simulačním kroku mají komponenty nedefinovaný signál. Týká se to konkrétně těchto signálů:

```
DC: ADDR
RF: ADDRa, ADDRb
IC: iADDR
ALU: ALU_op1, ALU_op2
```

Funkce `conv_integer`, která je použita pro konverzi typu std_logic_vector na typ integer, konvertuje nedefinované vstupní hodnoty na 0.

Blíže rozeberu varování týkající se jednotky ALU v druhém simulačním kroku.

0. simulační krok

Vstupy

```
CLK = 0b                        ADDRa = Xh
RESET = 1b                      ADDRb = Xh
CLK'event                       ALU_op1 = XXh
RESET'event                     ALU_op2 = XXh
iADDR = XXh                     RF_DO2 = XXh
INSTR = XXXXh                   imm_RF_sel = Xb
imm8 = XXh
```

Výstupy

PC:

```
iADDR = 00h
```

IC:

`INSTR = 7200h` (instrukce z adresy 00h, funkce conv_integer zkonvertovala nedefinovanou adresu na vstupu na 00h)

DEC:

Defaultní řídicí signály – `imm_RF_sel = 1`

```
ADDRa = Xh
ADDRb = Xh
imm8 = XXh
```

RF:

```
ALU_op1 = 00h
RF_DO2 = 00h
```

(načtené zdrojové operandy z adresy 00h – funkce conv_integer zkonvertovala nedefinované adresy na vstupu na 00h)

MX2-1

```
ALU_op2 = XXh
```

Pozn. multiplexor je implementován takto:

```
DO <= imm8 when imm_RF_sel = '1' else
      RF;
```

tedy pokud má řídicí signál neznámou hodnotu tak na výstup posílá operand z RF.

---

1. simulační krok

Vstupy

```
CLK = 0b                        ALU_op1 = 00h
RESET = 1b                      ALU_op2 = XXh
iADDR = 00h                     RF_DO2 = 00h
INSTR = 7200h                   ADDRa = Xh
imm_RF_sel = 1                  ADDRb = Xh
imm8 = XXh
```

Výstupy

Z pohledu PC, IC, RF nenastává na jejich výstupech žádná změna.

DEC:
`imm_RF_sel = 1` (zůstává na stejné hodnotě – dekódovaná instrukce je MOV rc, 0)
ADDRa = 2h
ADDRb = 0h
imm8 = 00h

MX2-1:
`ALU_op2 = XXh`
Tentokrát měl multiplexor řídicí signál imm_RF_sel nastaven na 1, a proto na výstup poslal hodnotu okamžitého operandu imm8.

---

2. simulační krok

Vstupy

`CLK = 0b`
`RESET = 1b`
`iADDR = 00h`
`INSTR = 7200h`
`imm_RF_sel = 1`
`imm8 = 00h`
`ALU_op1 = 00h`
`ALU_op2 = XXh`

Výstupy

Z pohledu PC, IC, DEC a RF nenastává žádná změna.

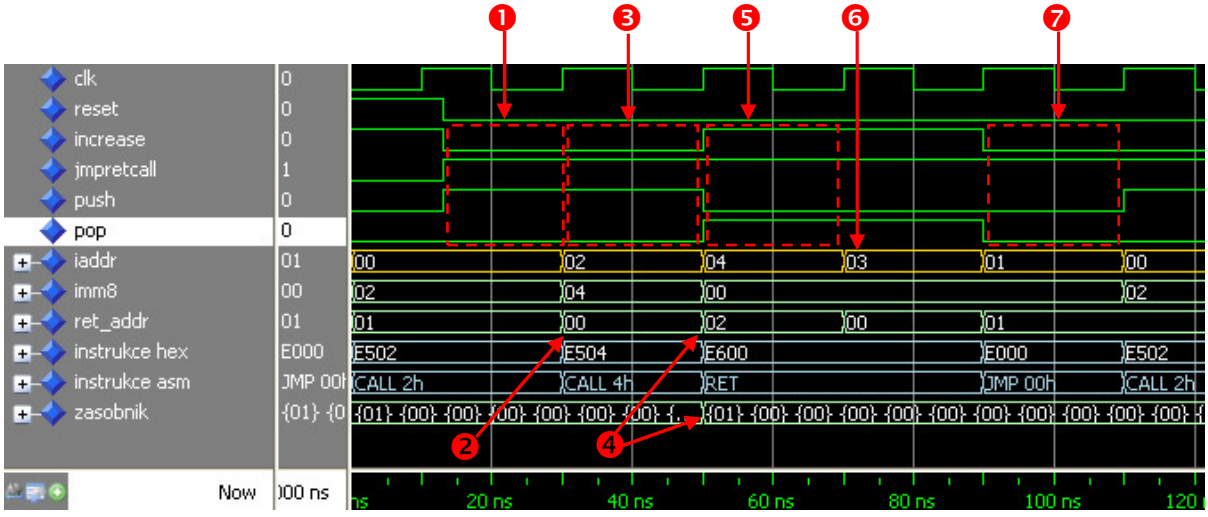
MX2-1:
`ALU_op2 = 00h`

Hodnota signálu ALU_op2 je tedy nastavena až na **konci** druhého simulačního kroku. Jednotka ALU tedy ještě ve druhém simulačním kroku počítá s nedefinovaným vstupem.

5.2.3.2 Eliminace řídicích konfliktů

Adresa hex	Instrukce hex	Význam instrukce
00	E502	call 2
01	E000	jmp 0
02	E504	call 4
03	E600	ret
04	E600	ret

Příklad demonstruje zanořené volání procedury a po ukončení procedur skok na začátek programu. Wave diagram s popisky můžeme vidět na obr. 24.



Obr. 24 – Wave diagram demonstrující eliminaci řídicích konfliktů

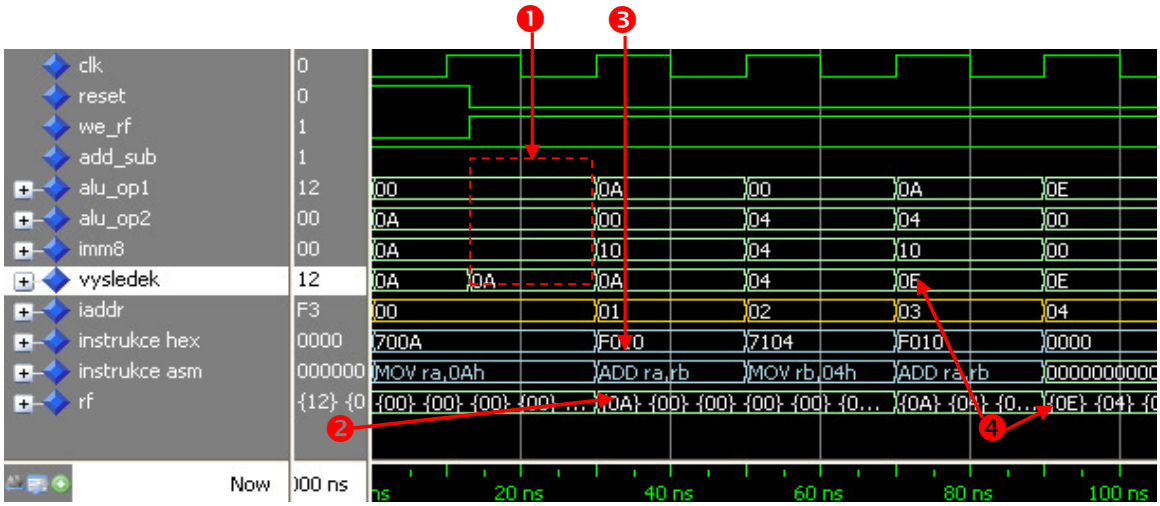
Jakmile je vynulován signál RESET, nastaví dekodér korektní řídicí signály pro instrukci `call 2` (❶). Je zrušen signál `increase` (PC nebude v příštím taktu inkrementován), nastaven signál `JmpRetCall` (do programového čítače bude nahrána adresa skoku – signál `imm8 = 2h`). Protože se jedná o instrukci volání procedury, je nastaven signál `push` do zásobníku návratových adres. Jako návratová adresa (signál `ret_addr`) je v tuto chvíli uvedena 01h. Takto je inicializováno dno zásobníku, ostatní položky v zásobníku jsou standardně nastaveny na 00h. Zásobník byl takto inicializován záměrně, aby bylo z diagramu zřejmé, že na výstupu ze zásobníku je vždy položka z adresy (ukazatel_do_zasobniku + 1) mod 256. Index do zásobníku ukazuje na první volnou položku. S nástupnou hranou CLK je do zásobníku zapsána návratová adresa, která je okamžitě vystavena jako návratová (❷). Dále je načtena a dekódována (❸) následující instrukce – v tomto případě `call 4`. Jedná se o stejnou instrukci jako v předchozím taktu, řídicí signály tedy zůstávají na stejných hodnotách. Změněna je pouze cílová adresa skoku (`imm8 = 4h`). S další nástupnou hranou CLK je opět uložena návratová adresa předchozí instrukce do zásobníku a vystavena na výstupu (❹). Dále je načtena a dekódována instrukce `ret` (❺). Programový čítač dostává korektní řídicí signály pro načtení následující instrukce – `increase = 1` (k adrese se bude přičítat jednička) a `JmpRetCall = 1` (jako vstupní adresa bude použita návratová adresa ze zásobníku – `Ret_ADDR = 02h`). Protože se jedná o instrukci návratu z procedury, je zrušen signál `push` a nastaven signál `pop`.

Programový čítač má tedy s následující nástupnou hranou všechny potřebné signály pro to, aby vystavil správnou adresu – 03h (❻). Na této adrese se nachází opět instrukce `ret` – řídicí signály tedy zůstanou na stejných hodnotách. Je inkrementován ukazatel do zásobníku návratových adres (aktuálně ukazuje na položku na adrese 254, na výstupu se objevuje položka z adresy 255 – adresa 00h). S následující hranou CLK vystaví PC adresu 01h, na které se nachází instrukce `JMP 0` (❼). Opět jsou nastaveny řídicí signály – `increase = 0` a `JmpRetCall = 1`. Programový čítač tedy vystaví s následující nástupnou hranou cílovou adresu skoku – 00h.

V takto implementované architektuře tedy nevznikají při zpracování řídicích instrukcí žádné pokuty. Programový čítač má vždy s nástupnou hranou hodinového signálu na vstupu korektní řídicí i datové vstupy.

5.2.3.3 Řešení konfliktu typu RAW

Adresa hex	Instrukce hex	Význam instrukce
00	700A	mov ra,FF
01	F010	add ra,rb
02	7104	mov rb,04
03	F010	add ra,rb



Obr. 25 Wave diagram demonstrující eliminaci konfliktu RAW

V prvním taktu je provedena instrukce `MOV ra,0A` (❶). Jelikož je zápis do RF synchronní, tak je hodnota do registru zapsána až s následující nástupnou hranou CLK (❷). S touto hranou už se však načítá následující instrukce `ADD ra,rb`, která potřebuje výsledek z předchozí instrukce (❸). Ke konfliktu zde nedochází. Z pohledu simulace je zápis proveden v prvním simulačním kroku, ve kterém jsou vstupy: `CLK = 1` a `CLK'event`. Požadavek z dekodéru na čtení tohoto operandu přichází však až na konci třetího simulačního kroku a je tedy zpracován až ve 4. simulačním kroku. Problém by nenastal ani kdyby přišel požadavek na čtení tohoto operandu dřív než by byl zapsán. Na začátku taktu by sice došlo k vypočtení chybného výstupu, ten by byl ale opraven v okamžiku, kdy došlo k zápisu výsledku z předchozího taktu do RF. Následuje instrukce `MOV rb,04` a za ní `ADD ra,rb`. Sčítají se tedy operandy 0A + 04. Opět je vidět, že na výstupu je správný výsledek, který je s následující nástupnou hranou CLK zapsán do RF (❹).

5.2.3.4 Vysledky syntézy

Implementovaný subskalární procesor bylo možné syntetizovat. Syntéza proběhla do FPGA architektury Spartan 3 velikosti XC3S200. Jako implementační jazyk byl zvolen VHDL, jako simulátor ModelSim PE.

Procesor může pracovat na frekvenci **58,333 MHz**.

Tabulka 7 zobrazuje výsledný report syntézy týkající se spotřeby funkčních bloků.
V tabulce 8 pak můžeme vidět skutečné výsledky získané z implementace procesoru.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	222	1920	11%
Number of Slice Flip Flops	15	3840	0%
Number of 4 input LUTs	462	3840	12%
Number of bonded IOBs	28	141	19%
Number of GCLKs	1	8	12%

Tabulka 7 – Odhad spotřeby funkčních bloků

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	15	3,840	1%	
Number of 4 input LUTs	456	3,840	11%	
Number of occupied Slices	234	1,920	12%	
Number of Slices containing only related logic	234	234	100%	
Number of Slices containing unrelated logic	0	234	0%	
Total Number of 4 input LUTs	456	3,840	11%	
Number used as logic	296			
Number used for Dual Port RAMs	32			
Number used for 32x1 RAMs	128			
Number of bonded IOBs	28	141	19%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	4.85			

Tabulka 8 – Skutečná spotřeba funkčních bloků

5.3 Návrh subskalární architektury s cache

Tato architektura vychází z dříve implementované subskalární architektury bez logiky IC a DC. Připojeny byly tedy logiky instrukční a datové cache. Návrh těchto komponent byl představen v kapitole 2.7 a nebude zde proto znovu popisován. Vyměněny byly tedy komponenty IC a DC. Přidána byla komponenta RAM. Ostatní části procesoru zůstali buď stejné nebo prošly pouze menšími změnami. Tyto změny jsou popsány níže.

Změněné komponenty:

IC, DC – zcela nová architektura komponent, je popsána v kapitole 2.7.

DEC – byla přidána instrukce NOOP a signál `RR_DC`. Tento signál je nastaven pouze v případě, že je zpracovávána instrukce LD. Adresa operandu do DC se totiž mění s každou instrukcí. To do teď nevadilo protože DC asynchronně vystavovala načítaná data, která stejně nepřešla přes multiplexor. Načítání z libovolných adres by ale nyní způsobovalo nechtěné výpadky z DC.

PC – byly zavedeny signály zastavující programový čítač při výpadku z IC nebo DC (signály `vypadek_IC, vypadek_DC`)

Nová komponenta:

RAM – paměť RAM, ze které jsou načítána data a instrukce do IC a DC. Architektura paměti je popsána v kapitole 2.7.

Celkem tedy přibyla pouze jedna komponenta. Architektura procesoru je nyní tvořena 15ti zdrojovými soubory.

5.3.1 Schéma procesoru

Schéma procesoru je už značně rozsáhlé proto bylo umístěno do přílohy.

5.3.2 Příklady

Následující příklady demosntrují funkčnost instrukční a datové cache. Wave diagramy simulace jsou již značně rozsáhlé. Vypuštěny proto byly všechny signály, které nesouvisí s aktuálně demonstrovanou ukázkou. Vnitřní signály IC a DC jsou v diagramech označeny fialově. Adresa právě prováděné instrukce je vždy zobrazena žlutě. Modře je zobrazena aktuálně prováděná instrukce – je zobrazen její operační kód i význam v assembleru [5].

Demonstrace algoritmů write back a LRU

Pro ukázku budu používat následující kombinace instrukcí

Adresa hex	Instrukce hex	Význam instrukce
00	9010	ld ra,16
01	9111	ld rb,17
02	7000	mov ra,0
03	A010	st 16,ra
04	9020	ld ra,32
05	7000	mov ra,0
06	A020	st 32,ra
07	9030	ld ra,48

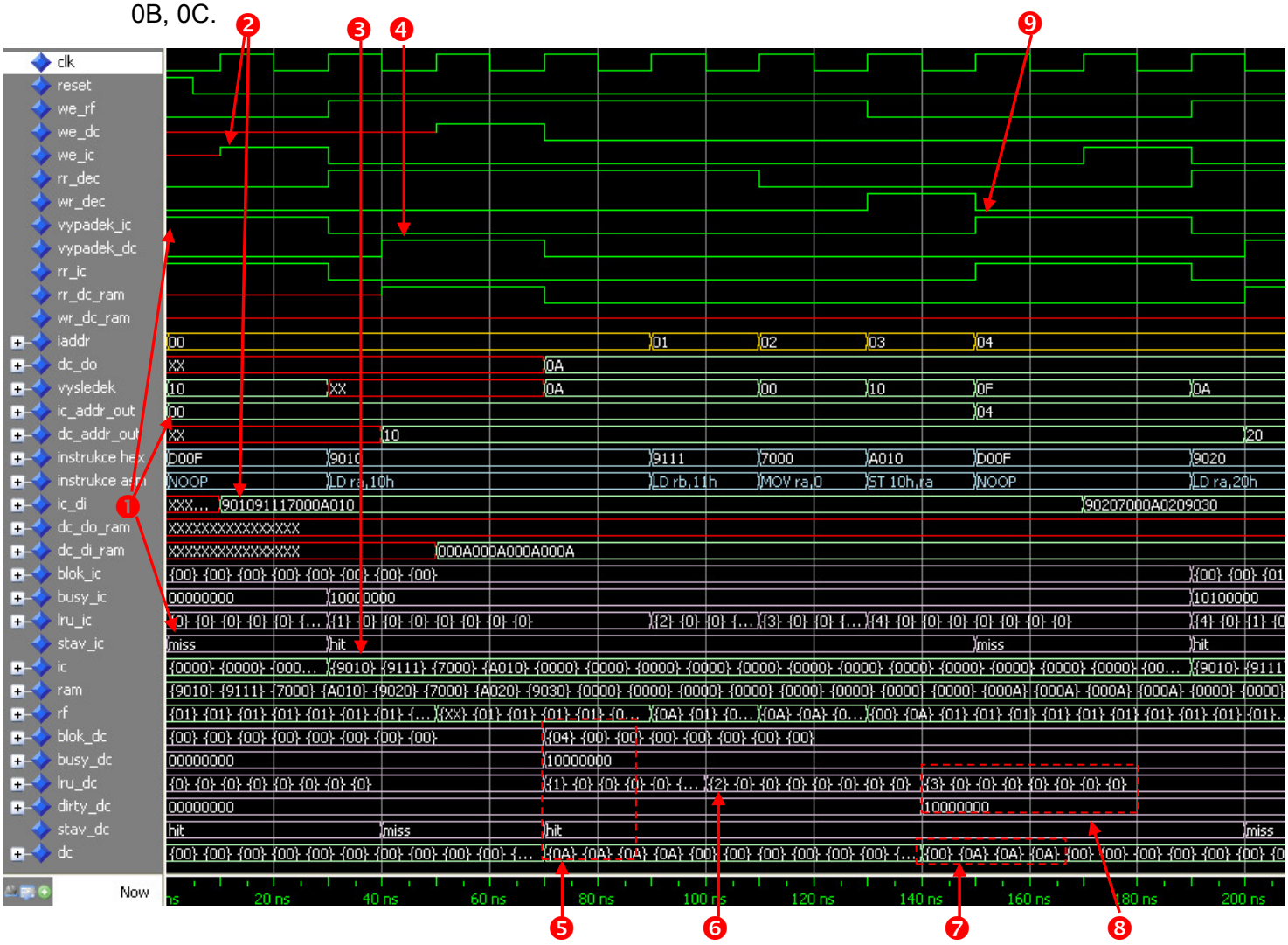
Do prvního setu datové cache načtu dva různé bloky z RAM, oba modifikuji a následně načtu do prvního setu jiný blok z RAM.

Nejprve načtu data z adresy 16dec (4. blok v paměti ram). Tento blok bude namapován na první set v DC a jelikož jsou oba bloky v prvním setu volné, umístí DC

načítaný blok na první pozici. Data budou přečtena a vystavena na výstup, čímž se inkrementuje čítač LRU. Následující instrukce načítá data ze stejného bloku. Data jsou přítomna v DC proto nenastane výpadek a pouze se inkrementuje čítač LRU u prvního bloku. Do registru ra uložím náhodná data, která pak přesunu zpět do načteného bloku v DC. To způsobí další inkrementaci čítače LRU a označení bloku dirty bitem. Počet přístupů do načteného bloku je v tuto chvíli 3.

Dále načtu nový blok z adresy 32dec. I tento blok je namapován na první set v DC. První blok v prvním setu je již obsazen, proto je tento blok umístěn do druhého bloku prvního setu. Data jsou vystavena na výstup. Následně je tento blok modifikován náhodnými daty. Je označen dirty bitem, hodnota čítače LRU je v tento okamžik 2.

Poslední instrukce načítá data z adresy 48dec. Tento blok je opět mapován na nultý set v DC. Oba bloky jsou však aktuálně zaplněné a navíc označeny dirty bitem. Datová cache tedy porovná LRU obou bloků a zjistí, že do druhého bloku bylo uskutečněno méně přístupů. Tento blok je proto odsunut zpět do RAM a na jeho místo je načten blok z adresy 48dec. Pozn. načítané bloky z paměti RAM jsou po řadě inicializovány na tyto hodnoty: 0A, 0B, 0C.

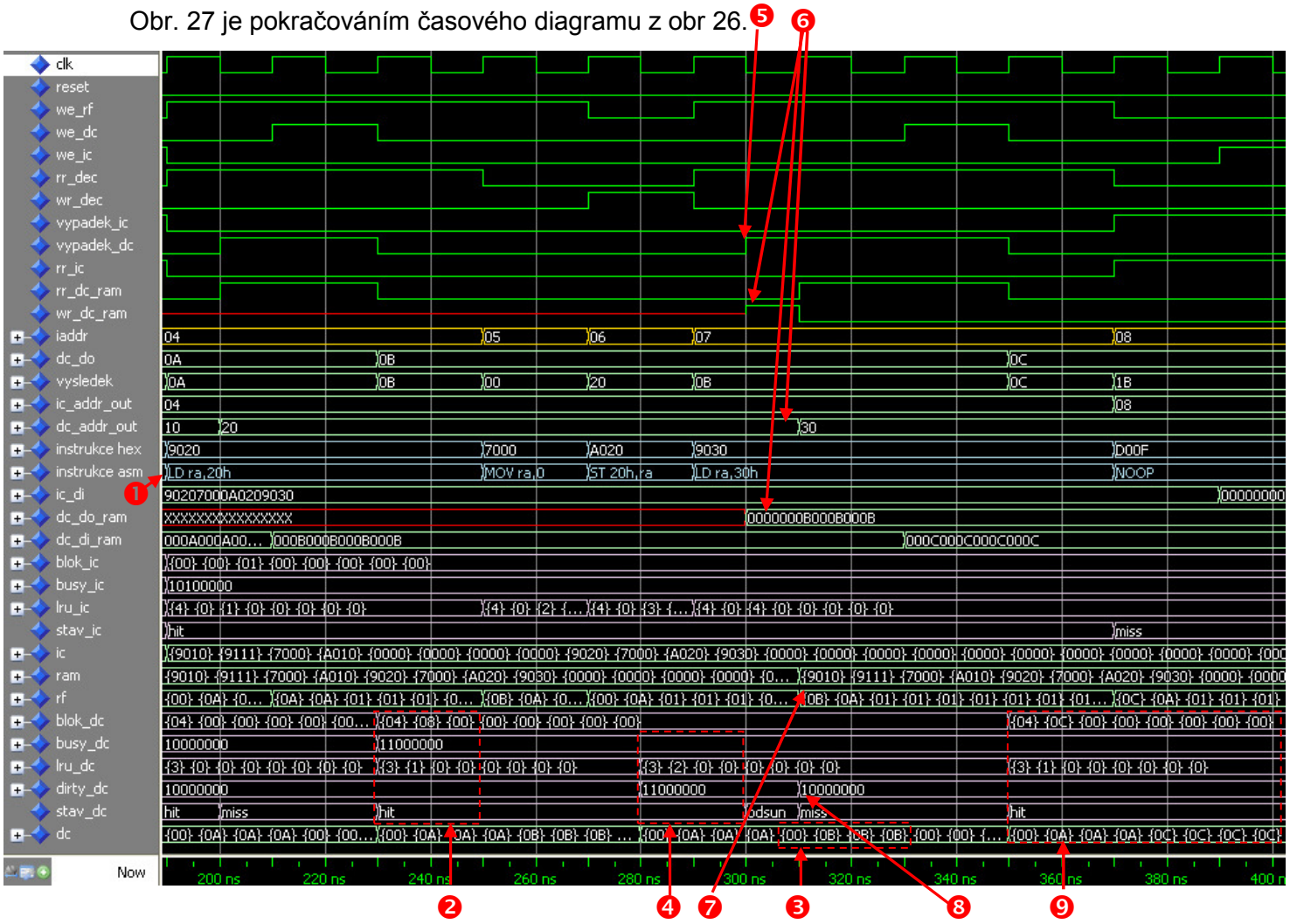


Obr. 26 - Wave diagram demonstrující funkčnost IC a DC 1 ze 2

Na počátku jsou instrukční i datová cache prázdné. S vystavením adresy do instrukční cache nastává první výpadek z IC. (Na adrese 00 se nachází instrukce LD ra,16). IC asynchronně nastaví svůj vnitřní stav na miss, nastaví signál `vypadek_ic` a vystaví adresu načítané instrukce do paměti RAM (❶). Při výpadku dále nastaví výstupní instrukci na NOOP. Význam tohoto kroku je demonstrován na následujícím příkladu. Paměť RAM

pracuje synchronně a proto s nástupnou hranou CLK vystaví data a nastaví signál `we_ic` indikující, že požadovaná data jsou připravena (❷). Velikost bloku je 4x2B, jsou tedy zaslány první 4 instrukce. S následující nástupnou hranou CLK jsou tyto instrukce zapsány do instrukční cache (❸), IC zruší požadavek na čtení z RAM (signál `rr_ic`), signál `vypadek_ic` nastaví na 0 a změní svůj vnitřní stav na *hit*. Paměť RAM zruší signál `we_ic`. IC si u načteného bloku nastaví busy bit, inkrementuje čítač přístupů LRU a poznamená si adresu bloku RAM. První blok instrukcí je tedy načten (vyřizování výpadku trvalo dva takty) do IC a může se tedy začít provádět první instrukce. Dedodér zjistí, že se jedná o instrukci LD, nastaví proto signál `rr_dec`. Datová cache je však v tuto chvíli ještě prázdná, nastává proto další výpadek, tentokrát z DC (❹). Ze synchronizačních důvodů začíná DC pracovat až se sestupnou hranou CLK. Výpadek je obsloužen obdobným způsobem jako výpadek z IC, trvá dva takty. Po zápisu dat do DC jsou aktualizována metadata u právě načteného bloku – adresa načteného bloku z RAM, nastaven busy bit, inkrementován čítač LRU a vnitřní stav změněn zpět na *hit*. (❺). Následující instrukce načítá data ze stejného bloku. Tato instrukce je použita pouze proto aby byl inkrementován čítač LRU (❻). První byte načteného bloku je modifikován. Obsah bloku se tedy změnil takto: `0A 0A 0A 0A -> 00 0A 0A 0A` (❼). Opět je inkrementován počet přístupů do bloku a daný blok je označen dirty bitem (❽). Při načítání následující instrukce nastane opět výpadek z IC (❾), který je obsloužen během dvou taktů tak jak bylo popsáno na začátku tohoto příkladu.

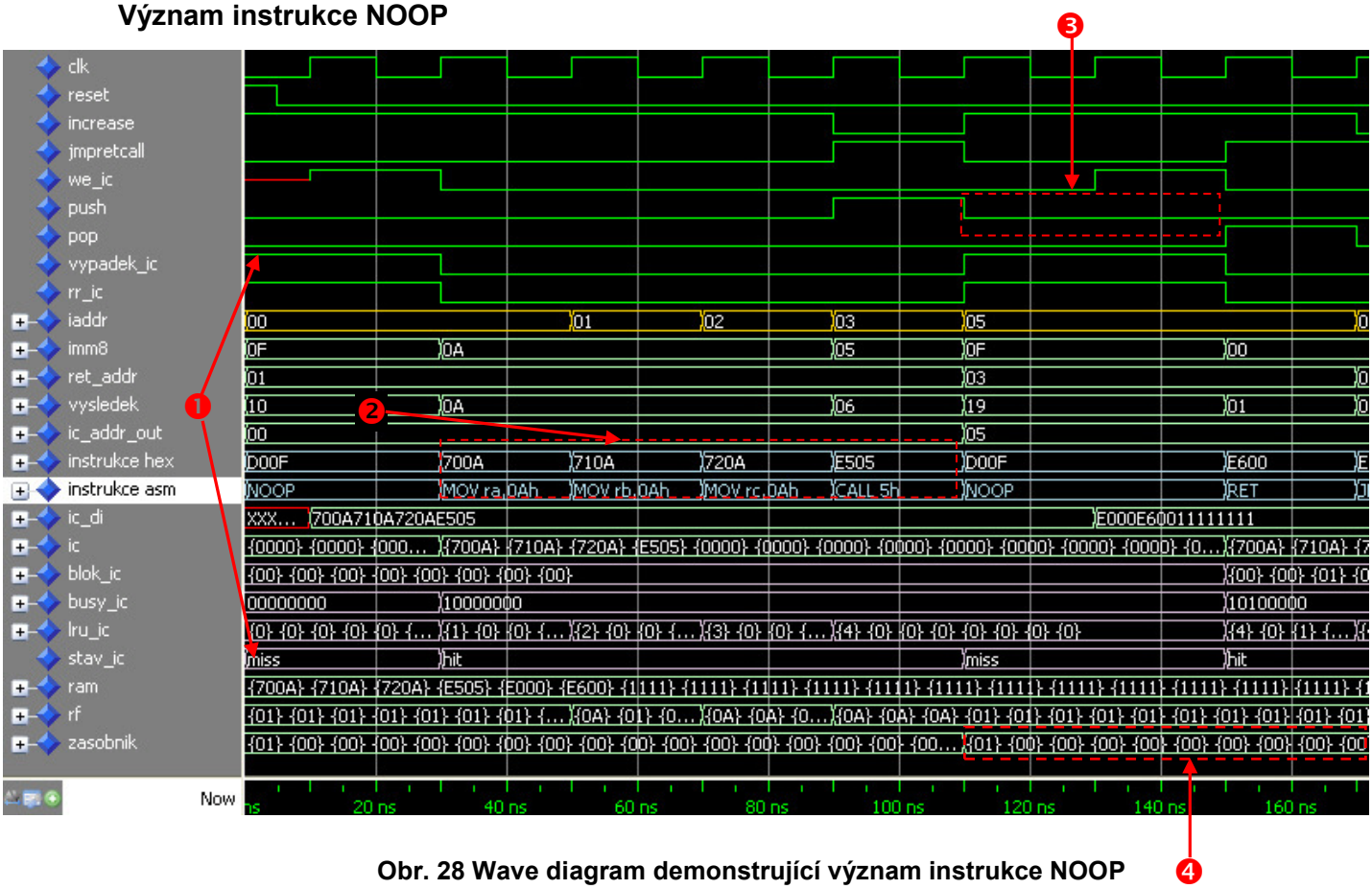
Obr. 27 je pokračováním časového diagramu z obr 26.



Obr. 27 Wave diagram demonstrující funkčnost IC a DC 2 ze 2

Načítaná instrukce byla `LD ra,20h` (❶). Při zpracování této instrukce opět dochází k výpadku z DC, který je obsloužen během dvou taktů. Data z adresy 20h jsou mapována na první set v DC, ve kterém je stále volný druhý blok. Data jsou proto umístěna tam (❷). Druhý načtený blok je následně modifikován instrukcí `MOV ra,0` takto: `0B 0B 0B 0B -> 00 0B 0B 0B` (❸). V prvním setu DC obsahují nyní oba bloky modifikované bloky RAM – dirty bity u obou bloků jsou nastavené. Do druhého bloku však byly učiněny pouze dva přístupy (❹). Následuje instrukce `LD ra,30h` a nastává další výpadek z DC. Blok začínající adresou 30h je opět mapován do prvního setu DC. Datová cache zjistí, že nemá načten požadovaný blok, proto nastaví signál `vypadek_DC` a změní svůj vnitřní stav na *miss*. Oba bloky v prvním setu DC jsou obsazené a proto se podle čítače LRU rozhodne, že nový blok z RAM bude zapsán do druhého bloku v cache – připomínám, že do druhého bloku byly pouze dva přístupy, do prvního tři přístupy. Druhý blok je však modifikován a musí být nejprve odsunut. Vnitřní stav je proto změněn na *odsun* (pozn. z diagramu není patrná změna stavů z *hit* na *miss* a dále na *odsun* protože tato změna je asynchronní). DC dále nastaví signál `wr_dc_ram`, adresu na kterou bude proveden zápis (adresa zůstává v tomto případě stejná protože odsouváme blok, který jsme při posledním výpadku načítali) a samotný blok dat (❻). Paměť RAM s nástupnou hranou CLK zapíše odsouvaný blok (❼). Vnitřní stav DC je změněn zpět na *miss*, je zrušen signál `wr_dc_ram` a u odsunutého bloku vynulován dirty bit (❽). Zatím ještě nedochází k vynulování čítače LRU. Následně začne být obsluhován výpadek stejným způsobem jako by byl přepisovaný blok prázdný. Odsunutý blok je tedy přepsán novým, je vynulován čítač LRU a aktualizována adresa načteného bloku (08 -> 0C) (❾).

Význam instrukce NOOP



Obr. 28 Wave diagram demonstrující význam instrukce NOOP

Pro demonstraci významu instrukce NOOP použiji následující příklad:

Adresa hex	Instrukce hex	Význam instrukce
00	700A	mov ra,0A
01	710A	mov rb,0A
02	720A	mov rc,0A
03	E505	call 5
04	E000	jmp 0
05	E600	ret

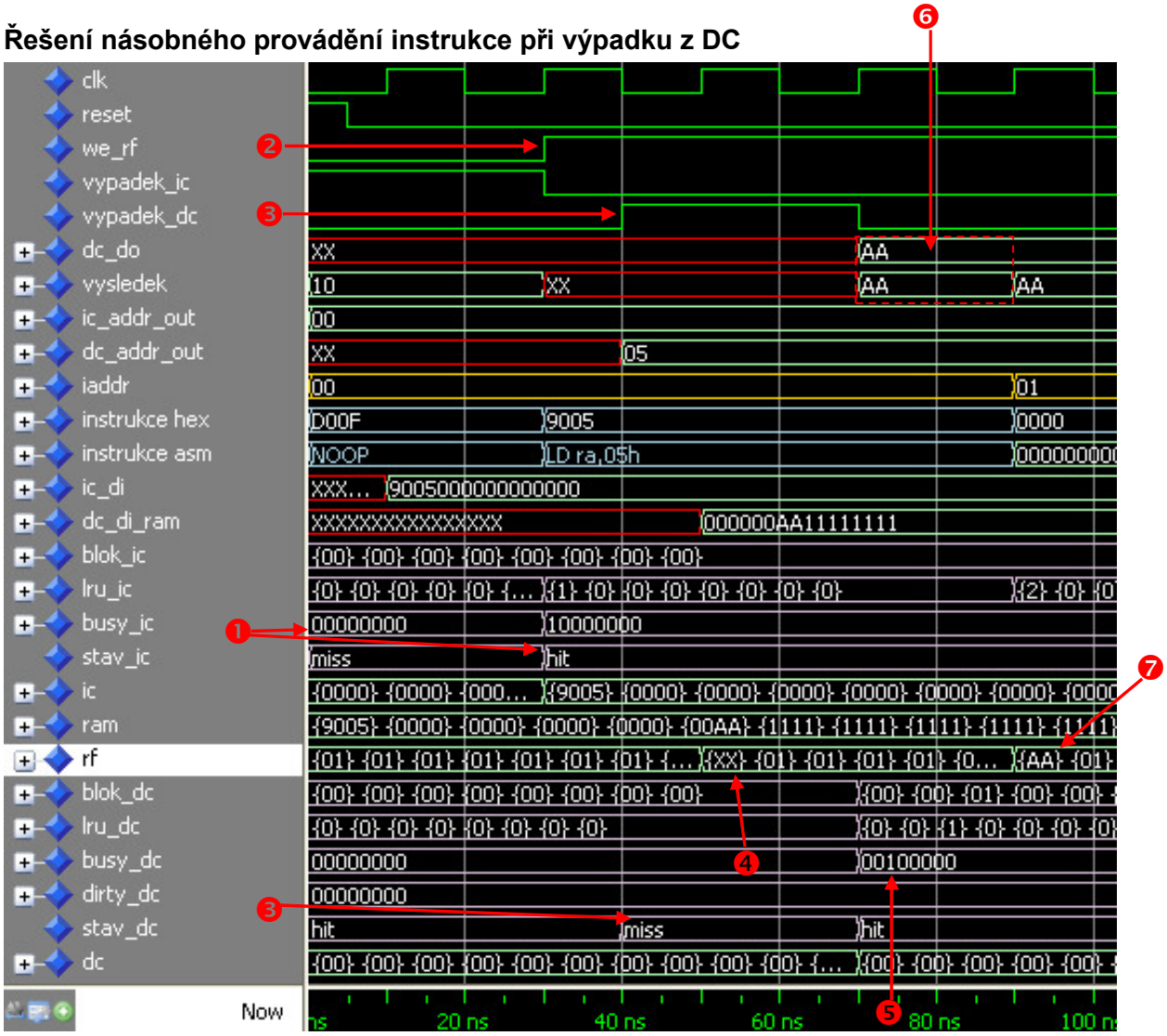
První tři instrukce budou pouze přesouvat data do registrů. Čtvrtá instrukce je volání procedury na adrese 05h. Tato adresa však odkazuje na instrukci mimo aktuálně načtený blok a proto nastane výpadek. Výpadek z IC trvá vždy dva takty. V případě výpadku z instrukční cache, vystaví IC na výstup instrukci NOOP. Dekodér tuto instrukci interpretuje tak, že nastaví standardní řídicí signály.

Pokud by IC nevystavila na výstup instrukci NOOP a na výstupu by po dobu výpadku zůstala instrukce CALL, ponechal by dekodér nastavený signál `push` po celou dobu výpadku – 2 takty navíc. Signál `push` by tedy byl vystaven celkem po dobu tří taktů a do zásobníku návratových adres by se uložila 1x adresa 03h (což je správně) a dále 2x adresa 05h (což je adresa vystavená programovým čítačem, ze které se aktuálně obsluhuje výpadek).

Nejedná se však pouze o problém pokud nastane výpadek po instrukci CALL. Problém by nastal u všech instrukcí, u kterých by vadilo, že jsou zpracovávané 3x za sebou na místo 1x. Jedná se tedy o aritmetické instrukce, posuvy a rotace, skoky. Ukázku vidíme na obr. 28.

První výpadek nastane při načítání první instrukce z IC (❶). V tuto chvíli by ještě nebylo nezbytně nutné aby instrukční cache vystavila instrukci NOOP. Dekodér by jednoduše dostal nedefinovaný vstup a k žádné „škodě“ by nedošlo. Po zpracování výpadku jsou provedeny první tři instrukce MOV a následuje instrukce CALL 5 (❷). Instrukce z adresy 05h však ještě není načtena a proto nastává znovu výpadek z IC. Dekodér obdrží instrukci NOOP, nastaví řídicí signály na standardní hodnoty – v tuto chvíli nás zajímá zejména signál `push`, který je nastaven na 0 (❸). Dole v diagramu můžeme pozorovat, že se obsah zásobníku během výpadku nijak neměni (❹).

Řešení násobného provádění instrukce při výpadku z DC



Obr. 29 Wave diagram činnost procesoru při výpadku DC

Příklad demonstruje provádění instrukce `LD ra,05h` – tedy načtení bytu na adrese 05h z paměti RAM do registru ra. Na dané pozici se nachází hodnota AAh.

Nejprve nastane výpadek z IC, který je během dvouch taktů obsloužen. První instrukce je načtena do IC a může být dekódována (❶). Dekodér rozpozná instrukci LD a mimo jiné povolí zápis do registrového pole (❷). Dekodér nemá žádnou informaci o tom jestli jsou data v DC nebo jestli nastal výpadek z DC. Data nejsou přítomna v DC a proto nastává výpadek z DC (ze synchronizačních důvodů až se sestupnou hranou CLK) (❸). Všimněme si teď výstupu z DC (signál `dc_do`) – výstup je stále nastaven na nedefinovanou hodnotu XX protože dosud nejsou v DC žádná data. Protože je však povolený zápis do RF, je tato nedefinovaná hodnota zapsána do registru ra s následující nástupnou hranou CLK (❹). Mezitím je obsluhován výpadek z DC, data jsou načtena z paměti RAM (data z adresy 05h jsou namapována do třetího bloku cache ❺). Teprv teď jsou vystavena korektní data na výstup z DC a dále přepnuta pomocí MX4-1 na signál `vysledek` (❻). S následující nástupnou hranou CLK jsou zapsána korektní data do registru ra (❼).

Pokud při zpracování instrukce LD nastane výpadek z DC, je do registrového pole dočasně zapsána chybná hodnota. Pokud se jedná o první výpadek z DC, je to nedefinovaná hodnota XX, pokud se jedná o druhý nebo další výpadek z DC, je to hodnota, která byla vystavena při posledním čtení/zápisu do DC. S načtením bloku do DC jsou vystavena na výstup korektní data a do RF je zapsána správná hodnota. Na rozdíl od výpadku z IC nevadí „násobné provedení“ instrukce LD při výpadku z DC.

5.4 Skalární architektura

Skalární architektura vystřídala subskalární architekturu. Základní princip je rozdělit logiku procesoru na několik stupňů. Instrukce se v daný takt nachází pouze v jednom z těchto stupňů a ostatní instrukce mohou být mezi tím zpracovávány v ostatních stupních. Rozdělením logiky procesoru na stupně je možné navíc dosáhnout vyšší frekvence. Ideálně, pokud je procesor rozdělen do n stupňů, je možné dosáhnout zrychlení téměř n-krát. S rozdělením procesoru na stupně vzniká ale nový problém – závislosti mezi instrukcemi. Tyto závislosti budou podrobně popsány dále v této kapitole. V implementační části bude podrobně popsáno jak jsem je vyřešil při implementaci různých verzí skalárního procesoru. Tabulka 9 znázorňuje princip fungování skalárního procesoru z pohledu stupňů procesoru a dále znázorňuje nástin problematiky závislostí typu RAW (Read After Write) a řídicích závislostí. Předpokládá se použití hardwarového předávání dat.

Uvažujme následující program:

```
MOV ra,80h
SL0 ra
JC 04h
SUB ra,1
MOV ra,3
ADD ra,5
```

Takt / Stupeň	1	2	3	4	5	6	7	8	9
IF	MOV	SL0	JC	SUB	SUB	MOV	ADD		
ID		MOV	SL0	JC →	JC →	JC	MOV	ADD	
EX			MOV	SL0	JC	JC	JC	MOV	ADD
WB				MOV	SL0	JC	JC	JC	MOV

Tabulka 9 – Zpracování instrukcí ve skalární architektuře

Procesor je v tomto případě rozdělen na 4 stupně – IF, ID, EX, WB. Zpracování prvních dvouch instrukcí probíhá tak jak bychom očekávali. Druhá instrukce je však závislá na výsledku první. Protože je však použito hardwarové předávání dat, je výsledek první instrukce předán ze stupně EX druhé instrukci do stupně ID. Výsledek je předán dříve než je zapsán do registru ra. Instrukce SL0 tedy ve skutečnosti nepracuje s registrem ra, ale pracuje s hodnotou, kterou obdržela ze stupně EX. Pokud by nebylo použito HW předávání dat, musela by instrukce SL0 čekat ve stupni ID jeden takt, než by byl výsledek z předchozí instrukce zapsán do registrového pole.

Třetí instrukce je instrukcí podmíněného skoku. Registr carry obsahuje hodnotu 1 a skok proto bude proveden. Hodnota do registru carry je však zapsána až v ve fázi WB (5. takt). Instrukce JC proto musí čekat ve stupni ID jeden takt než obdrží správnou hodnotu z registru carry. V tomto taktu musí vystavit korektní řídící signály pro programový čítač. S následující nástupnou hranou teprv PC načte správnou instrukci (MOV ra,3). A s další nástupnou hranou začne být tato instrukce dekódována. V tomto případě se tedy instrukce JC zdrží v dekódovací fázi tři takty. Existuje samozřejmě možnost použití obdobného triku jako bylo HW předávání dat. O možnostech použití této techniky bude podrobně mluveno v kapitole XY.

Skalární architektura nenabízí ještě „pravý“ paralelismus ve smyslu, že by bylo možné provádět několik instrukcí současně ve stejných stupních nezávisle na sobě. Jedná se pouze o tzv. časový paralelismus. Procesor potřebuje 4 takty pro dokončení první instrukce a pro každou další pouze jeden takt (pokud bychom neuvažovali skokové instrukce). Zjednodušeně by se dalo říct, že platí vztah:

$$\textit{Průměrné CPI} \approx 1$$

5.4.1 Typické stupně skalární architektury

IF…Instruction Fetch – načtení instrukci z instrukční cache v závislosti na adrese od programového čítače

ID…Instruction Decode – dekódování instrukce a načtení operandů. V tomto stupni jsou vystaveny všechny řídicí signály v závislosti na typu instrukce. Dále jsou načteny operandy z registrového pole.

EX…Execute – provádění instrukce. V závislosti na signálech nastavených dekodérem je instrukce provedena.

MA…Memory Access – přístup do datové cache. Tento stupeň je v mé architektuře sloučený se stupněm EX.

WB…Write Back – zápis výsledku do registru.

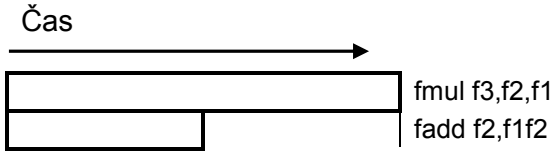
5.4.2 Závislosti mezi instrukcemi

a) Pravé

RAW…Read After Write – předchozí příklad byl ukázkou závislosti typu RAW, tedy závislosti kdy jedna instrukce potřebuje číst výsledek předchozí instrukce. Pokud je použito hardwarové předávání dat, neznamená tato závislost u takto implementované skalární architektury žádné zdržení.

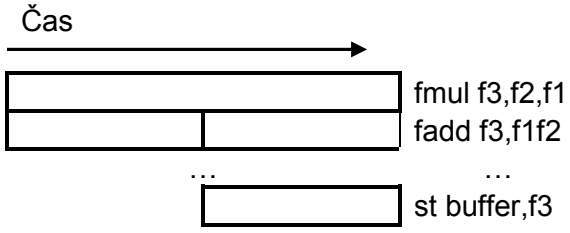
b) Nepravé

WAR …Write After Read – závislost kdy jedna instrukce chce zapsat svůj výsledek na místo, ze kterého ještě předchozí instrukce potřebuje číst.



Provedení instrukce `fmul` trvá daleko déle než provedení instrukce fadd. Mohlo by se tedy stát, že by instrukce `fadd` zapsala výsledek do registru f2 ještě v době kdy tento registr předchozí instrukce potřebuje pro čtení operandu.

WAW…Write After Write – závislost kdy jedna instrukce přepíše výsledek jiné instrukci.



Doba provádění instrukce `fmul` je opět značně větší než doba provádění `fadd`. Pokud budou tedy obě instrukce prováděny paralelně, `fmul` zapíše svůj výsledek do registru f3 po dokončení instrukce `fadd` a následně instrukce `st` uloží špatný výsledek.

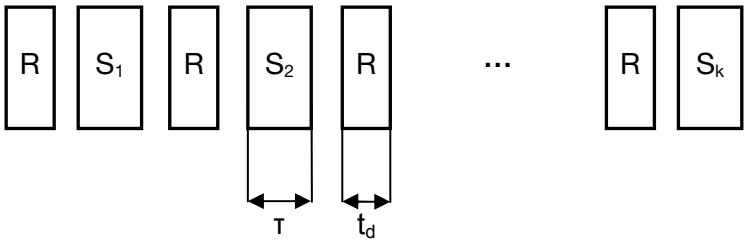
Konflikty typu WAR a WAW mohou nastat pouze u architektury, která zpracovává instrukce paralelně, jako např. superskalární architektura. S těmito konflikty se tedy u skalární architektury nesetkáme.

c) Řídicí

Jedná se o konflikty kdy skokové instrukce mění pořadí prováděných instrukcí. Rozpoznání skokové instrukce proběhne ve fázi dekódování instrukce (ID). Skoková instrukce postoupí s nástupnou hranou hodin do fáze ID. Ve stejný okamžik však programový čítač načte instrukci bezprostředně za skokovou instrukcí. Pokud bude skok proveden, je načtena špatná instrukce a načítání se musí opakovat.

5.4.3 Zrychlení subskalární architektury oproti skalární architektuře

Předpokládáme, že skalární architektura má k stupňů a k registrů pro uložení mezivýsledku, tak jak to vidíme na obr. 30.



Obr 30 – Rozdělení architektury do stupňů

Pokud uvažujeme, že při rozdělení celého modelu do *k* stupňů nevznikla žádná další režie, a že zpoždění všech stupňů je shodné, označíme zpoždění původní architektury jako *m*. Můžeme pak psát: m = k * τ. Kde τ je zpoždění jednoho stupně ve skalární architektuře. Zpoždění oddělovacího registru označíme jako t_d. Zrychlení při provádění N instrukcí pak můžeme vyjádřit takto:

$$S = \frac{T_1}{T_n} = \frac{N \cdot k \cdot \tau}{k(\tau + t_d) + (N-1) \cdot (\tau + t_d)}$$

První instrukci provedeme za $k*(\tau + t_d)$, každou další instrukci už ale dostaneme pouze za dobu $(\tau + t_d)$. Toto je ovšem pouze ideální případ kdy jsou instrukce po sobě jdoucí na sobě nezávislé a neuvažujeme skokové instrukce. Ve skutečnosti to ale neplatí (viz. diagram v tabulce 9), proto je nutné vypočítat průměrnou pokutu na jednu instrukci. Průměrná pokuta *q* říká kolik taktů navíc (díky instrukčním závislostem a skokům) bude trvat provedení jedné instrukce. Vzorec zrychlení pak bude vypadat takto:

$$S = \frac{T_1}{T_n} = \frac{N \cdot k \cdot \tau}{\left[k(\tau + t_d) + (N-1) \cdot (\tau + t_d)\right] * (1+q)}$$

5.5 Návrh skalární architektury bez cache

Návrh skalární architektury vycházel z návrhu subskalární architektury. Jedná se tedy opět o 8b load/store architekturu implementující instrukční sadu popsanou v kapitole 4. Většina komponent této architektury zůstala shodná se subskalární architekturou. Podrobně budou tedy popsány pouze ty komponenty, ve kterých nastaly zásadní změny. Subskalární architektura byla intuitivně rozdělena na 4 stupně. Rozdělením architektury na stupně bylo možné dosáhnout vyšší pracovní frekvence. Nově bylo potřeba řešit závislosti typu RAW a řídicí konflikty. Implementoval jsem tři verze skalárního procesoru bez logiky cache. Tyto verze se liší v řešení řídicích konfliktů, konkrétně ve zpracování podmíněných skoků. Celkové schéma procesoru znázorňuje obrázek Chyba! Nenalezen zdroj odkazů..

5.5.1 Popis částí procesoru

Programový čítač

Programový čítač je téměř totožný s tím, který byl implementován u subskalární architektury. Změna spočívá v zavedení nového signálu `PC_EN` povolujícího činnost čítače. Tento signál je důležitý při eliminaci řídicích konfliktů. Důvod jeho použití bude podbobně popsán v kapitole XY. Pokud je tento signál nastaven, je povolena činnost programového čítače.

Dekodér

Princip fungování dekodéru zůstal stejný jako u subskalární architektury. Implementován je tedy opět jako jeden velký case, který v závislosti na operačním kódu instrukce nastavuje řídicí signály, adresy zdrojových operandů a hodnotu cílového operandu. Rozdíl je ve zpracování skokových instrukcí. Část dekodéru zpracovávající tuto skkupinu instrukcí je implementována jako stavový automat. Skokové instrukce se totiž v dekódovací fázi zdrží dva nebo i tři takty. Kolikátý takt je už instrukce ve fázi ID je rozlišeno právě vnitřním stavem dekodéru. Aby byl možný synchronní přechod mezi jednotlivými stavy, je do komponenty zaveden hodinový signál.

Význam signálů

`WE_IR` – povoluje/zakazuje zápis do instrukčního registru

`PC_EN` – povoluje/zakazuje činnost programového čítače

Všechny ostatní signály zůstaly shodné.

Registrové pole

Změna u této komponenty spočívá v tom, že zápis výsledku již není řízen hodinovým signálem, ale nastavením signálu `WE_RF`. Pokud je tento signál nastaven, je povolen zápis. Tuto změnu bylo nutné udělat aby mohl zápis výsledku z fáze EX proběhnout hned v následujícím taktu. Po dokončení fáze EX je totiž s nástupnou hranou zapsán výsledek do výsledkového registru (VR). Pokud by byl RF synchronizován hodinovým signálem, proběhnul by zápis do RF až s následující nástupnou hranou. Fáze WB by tedy trvala dva takty.

Jiné možné řešení tohoto problému by bylo odstranění výsledkového registru. Zápis by pak proběhnul s následující nástupnou hranou CLK, tak jak je očekáváno. Při návrhu procesoru jsem zvolil ponechání VR a „synchronizování“ zápisu do RF pomocí signálu `WE_IR`.

Registr

Před zásobník návratových adres musel být umístěn registr, který s každou nástupnou hranou CLK zaznamená adresu vystavenou programovým čítačem. Pokud by totiž právě načítaná instrukce byla instrukcí volání procedury – `call`, byla by rozpoznána až v dalším

taktu (ve fázi ID). V tento okamžik by už však PC vystavil adresu nové instrukce a ta by byla zapsána do zásobníku.

Komparátory adres

Tyto komparátory byly použity pro hardwarové předávání dat. Komparátory porovnávají adresu výsledku (přivedenou z fáze EX) s adresou zdrojových operandů. Pokud se obě adresy shodují, znamená to, že aktuální instrukce potřebuje operand, který je výsledkem předchozí instrukce (dochází ke konfliktu RAW). V takovém případě vyšle komparátor signál pro multiplexor, který přepíná mezi operandem z RF a operandem, který je výsledkem předchozí instrukce.

Instrukční registr (IR)

Registr umístěný za instrukční cache. Má šířku 16b a je do něj načtena instrukce z IC.

Datový registr (DR)

Registr umístěný za dekodérem a registrovým polem. Tento registr má šířku 38b a ukládají se do něj zdrojové operandy, adresa výsledku a veškeré řídicí signály z dekodéru, které jsou nutné pro provedení instrukce ve fázích EX a WB.

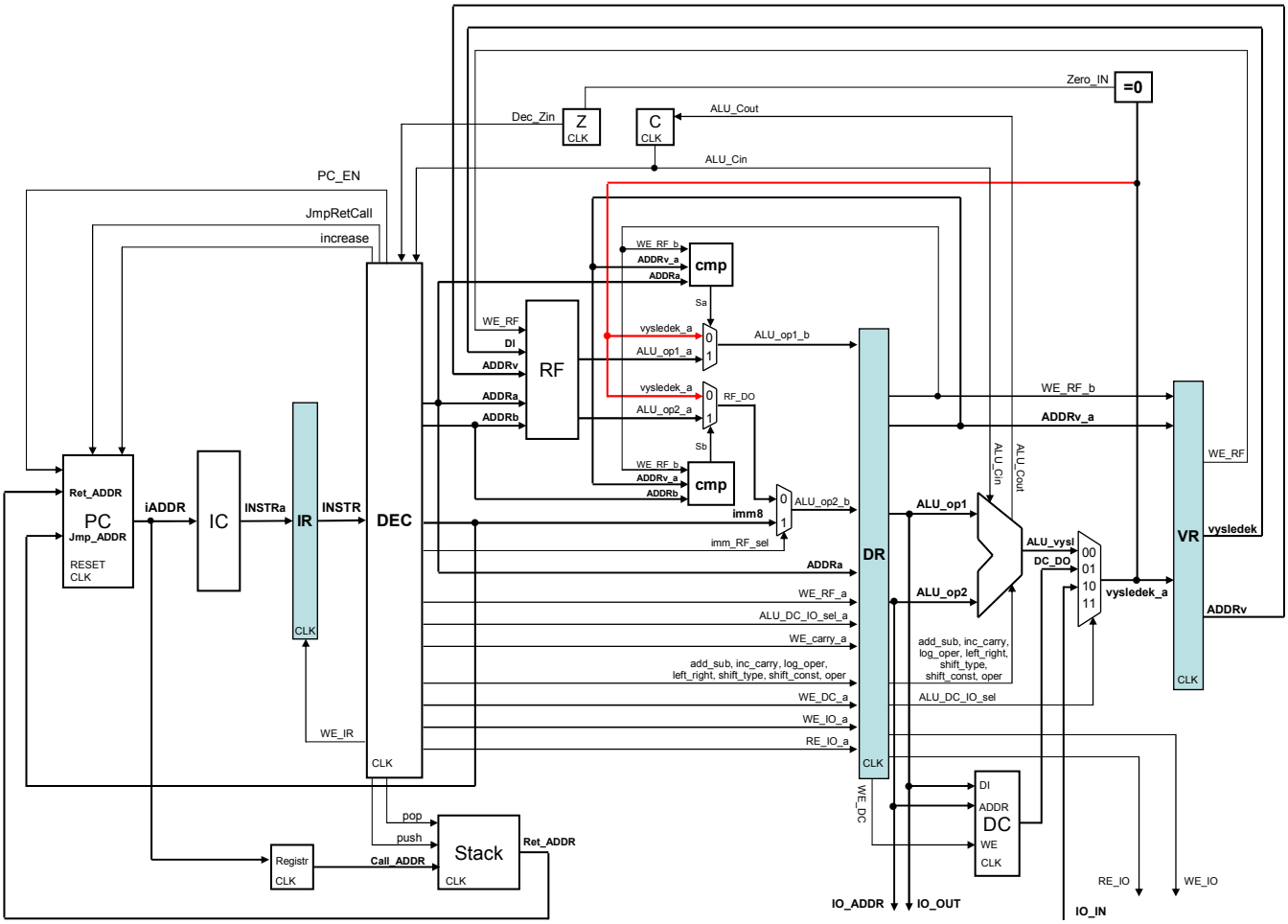
Výsledkový registr (VR)

Registr umístěný za jednotkou ALU. Do tohoto registru je uložen výsledek, adresa výsledku a signál `WE_RF` signalizující, zda je povolený zápis do RF.

Komponenty IC, DC, zásobník návratových adres, ALU, registry carry a zero, komparátor na nulu a multiplexory nebylo nutné měnit.

Schéma a zapojení jednotlivých komponent je vidět na obrázku Chyba! Nenalezen zdroj odkazů.. Červeně je znázorněno HW předávání dat.

Protože jsou některé signály „kopírovány“ mezi jednotlivými stupni architektury, snažil jsem se při pojmenování signálů dodržovat určitou konvenci. Dekodér například nastavuje signál `WE_RF`, který povoluje zápis do registrového pole. Tento signál se ve fázi ID jmenuje `WE_RF_a`, ve fázi EX `WE_RF_b` a ve fázi WB `WE_RF`. Signál má tedy své „originální“ jméno v místě, kde vstupuje do jednotky pro kterou je určen. Před touto jednotkou má za posledním podtržítkem písmeno označující o kolikátou kopii daného signálu se jedná.



Obr. 31 – Schéma skalární architektury

5.5.2 Stručný popis funkčnosti

Programový čítač je nulován signálem `RESET` a jeho činnost je povolována signálem `PC_EN`. Pokud není signál `RESET` nastaven a je povolena činnost PC, tak s každou nástupnou hranou CLK vystaví na základě řídicích signálů adresu instrukce. IC z této adresy následně asynchronně načte instrukci. Toto je činnost fáze IF.

S nástupnou hranou CLK je vystavená instrukce zapsána do instrukčního registru (IR) a předána na výstup dekodéru. Za předpokladu, že se nejedná o skokovou instrukci, vystaví dekodér veškeré řídicí signály, adresy zdrojových operandů a okamžitý operand. Zdrojové operandy jsou načteny z registrového pole. Adresy zdrojových operandů jsou porovnány s adresou výsledku předchozí instrukce a pokud jsou shodné a předchozí instrukce má povolený zápis výsledku do RF, tak je jako zdrojový operand aktuální instrukce použit výsledek z předchozí instrukce. Toto byla fáze ID.

S nástupnou hranou CLK jsou všechny potřebné řídicí signály, zdrojové operandy a adresa výsledku uloženy do datového registru (DR). Následně je provedena aritmeticko-logická operace, načtení/uložení výsledku z/do DC nebo externího zařízení. Toto byla fáze EX.

S další nástupnou hranou je zapsán výsledek, adresa výsledku a hodnota signálu `WE_RF` do výsledkového registru (VR). Pokud byl signál `WE_RF` nastaven, tak je výsledek zapsán do registrového pole. Toto byla fáze WB.

5.5.3 Řešení konfliktů

Konflikt typu RAW

Tento konflikt nastává v případě, že instrukce, která je aktuálně ve fázi ID potřebuje jako jeden ze svých zdrojových operandů výsledek předchozí instrukce. Tedy např.

```
add ra,8
add ra,5
```

Aby druhá instrukce nemusela čekat ve fázi ID než předchozí zapíše výsledek do registru, je výsledek z fáze EX vyveden do multiplexorů fáze ID (tento signál je na obrázku Chyba! Nenalezen zdroj odkazů. znázorněn červeně). Z datového registru jsou dále vyvedeny signály `WE_RF` a `ADDRv`. Pokud se shodují adresy výsledku (`ADDRv`) s adresou jednoho ze zdrojových operandů (`ADDRa`, `ADDRb`) a je povolen zápis do registrového pole, je nastaven signál `Sa` nebo `Sb` na hodnotu 0. Jako zdrojový operand je poté použit výsledek předchozí instrukce.

Signál `WE_RF` zavedený do komparátorů adres je klíčový. Jeho význam ukážu na následujícím příkladu.

Mějme instrukce:

```
JMP AAh         1110000010101010
ADD ra,05       0000000000000101
```

Červený rámeček ukazuje pozici na které je adresa prvního zdrojového operandu. I když instrukce JMP není aretmeticko-logická, stejně musí projít všemi stupni procesoru a dostane se tedy i do stupně EX. Pokud by nebyl do komparátoru zaveden signál `WE_RF`, komparátor by vyhodnotil obě adresy jako shodné a instrukci ADD by místo obsahu registru `ra` dal jako vstupní operand výsledek fáze EX, ve které se momentálně nachází instrukce JMP. Instrukce ADD by tedy dostala na vstup chybné vstupní operandy.

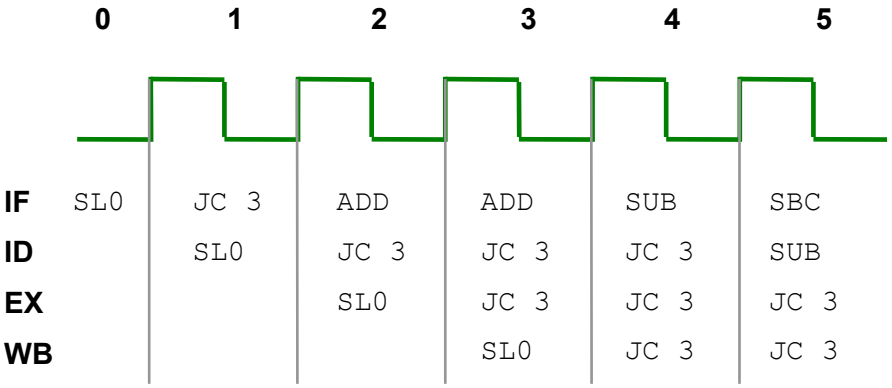
Řídicí konflikty

Tyto konflikty je možné řešit několika způsoby. Postupně jsem vytvořil tři verze skalární architektury, které se od sebe liší řešením řídicích konfliktů při zpracování podmíněných skoků. Tyto verze jsou na CD označené jako 1.0, 1.1 a 1.2.

Činnost všech tří procesorů bude demonstrována na následující skupině instrukcí.

```
SL0 ra
JC 3
ADD ra,1
SUB ra,1
SBC ra,1
```

Průchod jednotlivých instrukcí v implementaci verze 1.0 je znázorněn v diagramu na obrázku 32. Předpokládejme, že registr carry obsahuje hodnotu 0 a registr ra hodnotu 10000000b.



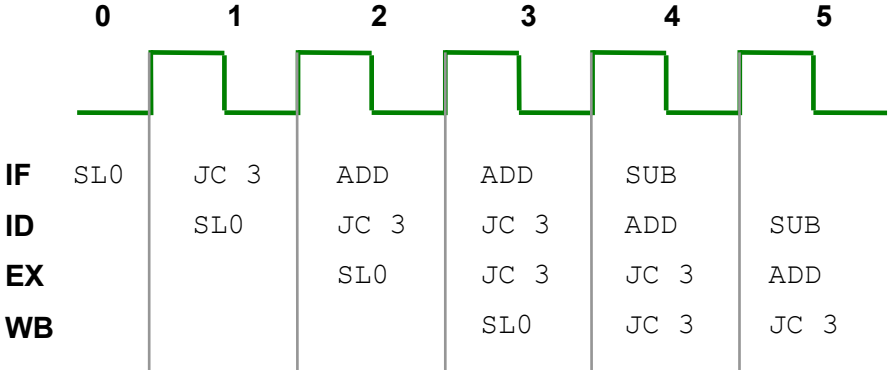
Obr. 32 – Řídicí konflikty při pokutě 2 takty za každý podmíněný skok – implementace verze 1.0

Ve druhém taktu je dekódována instrukce JC 3. V závislosti na obsahu registru carry bude nebo nebude proveden skok. Zápis do registru carry je synchronní a je prováděn až ve fázi WB. Připomínám, že registr carry je na začátku inicializován na nulu. Kdyby tedy měl dekodér v tento okamžik rozhodnout o tom jestli bude nebo nebude proveden skok, skok by se neuskutečnil. Dekodér proto musí pro tuto chvíli zastavit chod programového čítače (PC_EN = 0), zakázat zápis do instrukčního registru (WE_IR = 0) a čekat na zápis instrukce SL0 do registru carry. Ten bude proveden ve třetím taktu. Ve 3. taktu už má dekodér dostupný korektní vstup z registru carry. Ten má teď hodnotu 1 a skok tedy bude proveden. Dekodér vystaví korektní řídicí signály pro PC a povolí jeho činnost (PC_EN = 1). S nástupnou hranou 4. taktu vystaví PC korektní instrukci – SUB. Dekodér povolí zápis do IR (WE_IR = 1). V tomto taktu jsou také nastaveny signály increase = 1 a JmpRetCall = 0, což jsou hodnoty odpovídající neskokovým instrukcím. S nástupnou hranou 5. taktu je načtena instrukce SBC. Instrukce SUB je zapsána do IR a dekódována.

V této implementaci tedy trvají všechny podmíněné skoky 3 takty.

Nepodmíněné skoky trvají pouze dva takty. Není potřeba čekat na výsledky z registrů carry nebo zero. V prvním taktu dekódovací fáze jsou vystaveny korektní řídicí signály pro PC a je zakázán zápis do IR. V druhém taktu je povolen zápis do IR a nastaveny signály increase a JmpRetCall na standardní hodnoty.

Diagram na obrázku 33. znázorňuje zpracování podmíněných skoků pokud skok nenastane. Tato verze je implementována ve skalární architektuře verze 1.1. Následující příklad předpokládá obsah registru ra inicializovaný na hodnotu 00000000b.

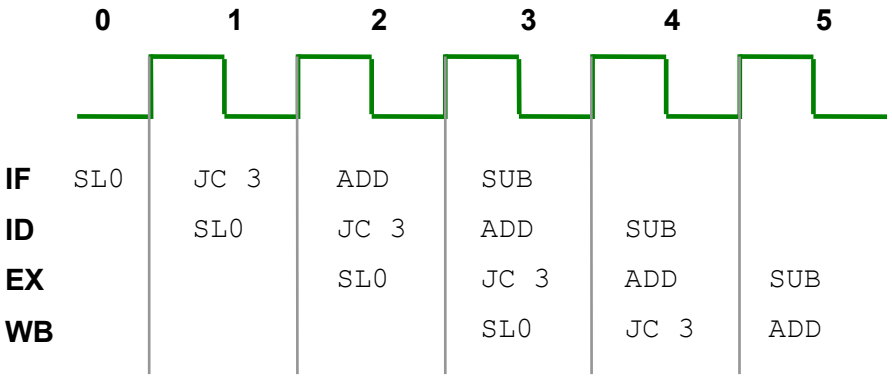


Obr. 33 – Řídicí konflikty při pokutě pouze 1 takt za podmíněný skok, který se neprovede – implementace verze 1.1

Verze 1.1 je vylepšením předchozí verze a zohledňuje fakt, že pokud podmíněný skok nenastane, tak programový čítač již načetl korektní instrukci v předchozím taktu. Není tedy důvod aby v takovém případě čekala instrukce JC v dekodéru další takt (tak jako čeká tato instrukce pokud má být skok proveden). Ve 3. taktu tedy opět dostávám korektní vstup z registru carry. Zjišťuji, že se skok nekoná. Následující instrukce je tedy instrukce ADD – tato instrukce je však již načtena. Povolím tedy činnost PC, zápis do IR a signály increase a JmpRetCall nastavím na standardní hodnoty. S nástupnou hranou 4. taktu je instrukce ADD uložena do IR a dekódována.

Podmíněné skoky jsou v této verzi prováděny 2 takty pokud se skok neprovede a 3 takty pokud se skok provede. Je tedy zavedena jakási negativní predikce skoku.

Diagram na obrázku 34 znázorňuje zpracování podmíněných skoků, které nejsou provedeny. Tato varianta je implementována verzí procesoru 1.2 a zavádí obdobu hardwarového předávání dat pro registry carry a zero. Následující příklad předpokládá obsah registru ra inicializovaný na hodnotu 00000000b.



Obr. 34 – Řídicí konflikty při pokutě 0 taktů za podmíněný skok, který se neprovede – implementace verze 1.2

Verze 1.2 využívá myšlenku zavedení hardwarového předávání dat také u registrů carry a zero. Pro tuto verzi musela být mírně pozměněna architektura procesoru. Schéma této verze je možné najít v přílohách. Do dekodéru teď nejsou zavedeny výstupy z registrů carry a zero, ale přímo signály z jednotky ALU a z komparátoru na nulu. Už ve druhém taktu tedy dostává dekodér aktuální signály carry a zero a může rozhodnout zda bude skok proveden či nikoli. Obrázek 34 znázorňuje situaci kdy skok nebyl proveden. V takovém případě nevzniká žádná pokuta a linka se nezastavuje. V případě, že by byl skok proveden, by se muselo čekat jeden takt na vystavení nové instrukce.

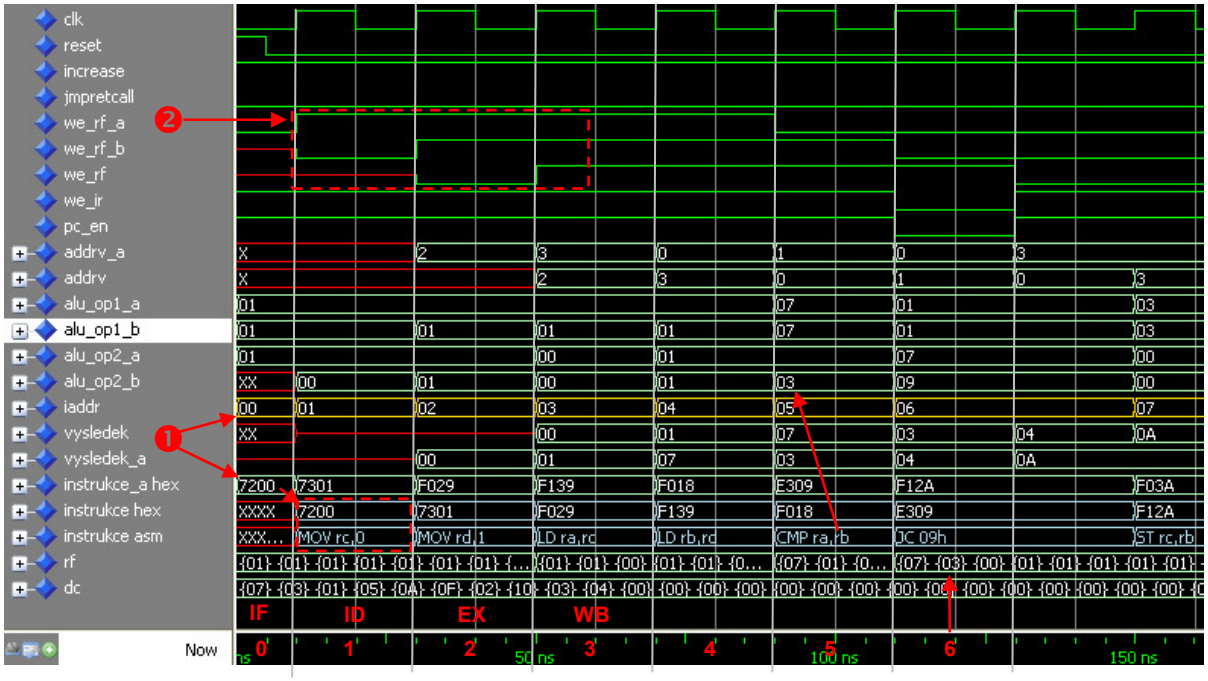
Tato varianta tedy opět snižuje pokutu u podmíněných skoků. **Pokud není skok uskutečněn, trvá provedení instrukce JC jeden takt**, pokud se skákat bude, trvá provedení JC dva takty. Tento způsob je nejvýhodnější z hlediska počtu taktů. Zavedením zpětné vazby se však prodlužuje kritická cesta což snižuje frekvenci procesoru. Ve výsledku není tato varianta rychlejší než ty předchozí.

5.5.4 Příklady

Ve všech příkladech, které budou následovat je použita verze architektury 1.1, tedy verze s nejmenší pokutou za podmíněné skoky a zároveň nejvyšší frekvencí.

5.5.4.1 Bubble sort

Obrázek 35 zobrazuje zpracování několika prvních instrukcí algoritmu bubble sort. Algoritmus i nastavení datové cache je shodné s tím, které bylo prezentováno v kapitole 5.2.3.1. Na snímku je dobře vidět jak instrukce postupně prochází jednotlivými stupni procesoru.



Obr. 35 – Algoritmus bubble sort na skalární architektuře

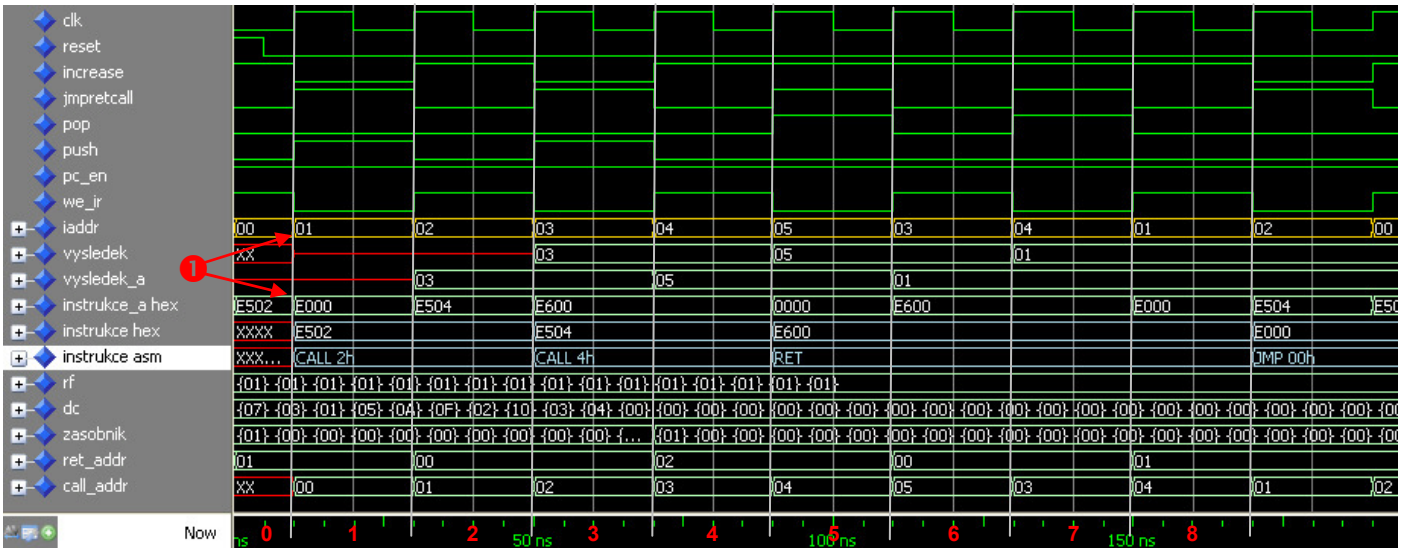
Na počátku je nastaven signál `RESET`, který nuluje programový čítač. Je tedy vystavena adresa 00h a asynchronně načtena první instrukce (❶). Jedná se o fázi IF. S nástupnou hranou `CLK` je instrukce zapsána do registru IR a dekódována. Protože se jedná o instrukci MOV, tak je povolen zápis do registrového pole (WE_RF = 1). Z diagramu je dobře vidět jak nastavení tohoto signálu postupně prochází jednotlivými stupni procesoru (ID, EX, WB) (❷). Ve spodní části obrázku je zaznačeno v jakých fázích se postupně nachází instrukce MOV rc,0. Přítomnost této instrukce ve fázi EX je možné rozpoznat podle signálu `vysledek_a`, ve fázi WB pak podle signálu `vysledek`.

Z diagramu je dále vidět řešení konfliktu RAW. Instrukce `LD, rb,rd` načítá hodnotu do registru rb. Tato hodnota je do registru rb zapsána až v 6. taktu. Instrukce `CMP ra,rb` však už v 5. taktu potřebuje tuto hodnotu načíst aby mohla v 6. taktu provést porovnání. V 5. taktu můžeme vidět, že díky HW předávání dat má už tuto hodnotu k dispozici (ALU_op2_b = 03h).

Procesor potřebuje 597 taktů pro seřazení vstupní řady 10ti čísel.

5.5.4.2 Volání procedur a nepodmíněné skoky

Následující diagram zobrazuje řízení procesoru při zanořeném volání procedur a zpracování nepodmíněných skoků. Zoubor zpracovávaných instrukcí je opět totižný s tím, který byl představen v kapitole 5.2.3.2



Obr. 36 – Ukázka zpracování nepodmíněných skoků

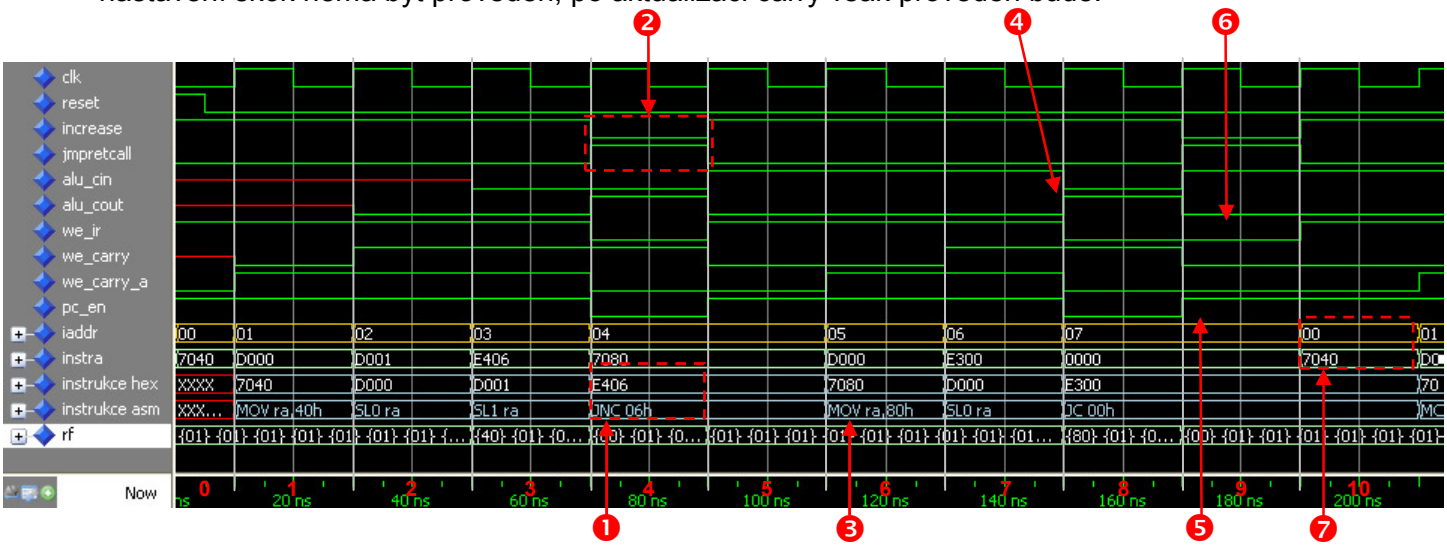
Diagram znázorňuje funkčnost procesoru při provádění nepodmíněných skoků. Na počátku je načtena instrukce CALL. V prvním taktu je dekódována. Programový čítač mezitím vystavil následující adresu a byla načtena následující instrukce E000 (❶). Protože se jedná o instrukci CALL byl také nastaven signál `push`. Ve druhém taktu už má PC na vstupu korektní řídicí signály i správnou adresu skoku a proto vystaví správně adresu 02h. Je načtena nová instrukce (E504). Dekodér v tomto taktu povolí zápis do IR (signál `WE_IR`) a nastaví signály `increase` a `JmpRetCall` na standardní hodnoty. Do zásobníku je uložena návratová adresa první instrukce a je vystavena na výstup (signál `Ret_ADDR`). Ve třetím taktu je dekódována nová instrukce – jedná se opět o instrukci CALL. Celý postup se opakuje, provádění trvá opět dva takty. V 5. taktu je dekódována instrukce RET. Její zpracování probíhá podobně jako u instrukce CALL s tím rozdílem, že jsou vystavené jiné řídicí signály a je nastaven signál `pop`. V 6. taktu tedy PC vystaví návratovou adresu ze vstupu `Ret_ADDR + 1`, je načtena instrukce z adresy 03h. Z diagramu je vidět, že instrukce RET strávila v dekódovací fázi celkem 4 takty. Je to proto, že se jedná o návrat ze zanořené procedury. Jdou tedy dvě instrukce RET za sebou. Z diagramu je dále vidět, že nepodmíněné skoky trvají vždy dva takty – v prvním taktu jsou dekódovány a nastaveny řídicí signály pro skokové instrukce, ve druhém taktu je načítána instrukce z cílové adresy skoku a dekodér nastavuje řídicí signály na standardní hodnoty.

5.5.4.3 Podmíněné skoky

Zpracování podmíněných skoků uvedu na následujícím příkladě:

Adresa hex	Instrukce hex	Význam instrukce
00	7040	mov ra,40h
01	D000	sl0 ra
02	D001	sl1 ra
03	E406	jnc 06h
04	7080	mov ra,80h
05	D000	sl0 ra
06	E300	jc 00h

Do registru ra nejprve nakopíruju hodnotu 01000000b, kterou poté posunu doleva. Po provedení instrukce SL0 ra bude v registru carry hodnota 0. Registr ra poté opět posunu doleva (instrukce SL1 je použita pouze proto aby bylo z diagramu dobře poznat, že se jedná o odlišnou instrukci od té předchozí). Po provedení této instrukce bude registr carry obsahovat hodnotu 1. Následuje instrukce JNC. Skok nebude proveden, nicméně při prvním dekódování této instrukce budou řídicí signály pro PC nastaveny tak jako by skok měl být proveden. Je to protože v tomto okamžiku ještě není registr carry nastaven na hodnotu 1. Signály jsou korigovány ve druhém taktu dekódování, kdy už je k dispozici aktuální hodnota registru carry. Poslední tři instrukce slouží k demonstraci opačné situace – podle úvodního nastavení skok nemá být proveden, po aktualizaci carry však proveden bude.



Obr. 37 – Ukázka zpracování podmíněných skoků

Ve druhém taktu můžeme vidět, že je poprvé nastaven signál `ALU_Cout` na hodnotu 0. Je to protože je v tomto taktu zpracovávána instrukce `MOV ra,40h` a tato instrukce nastavuje carry na nulu. Zápis této hodnoty co carry je vidět ve 3. taktu (signál `ALU_Cin`). Pozn. signál `ALU_Cin` slouží jako výstup z carry do jednotky ALU i do dekodéru – podle tohoto signálu se tedy rozhoduje o podmíněných skocích JC a JNC. Ve třetím taktu je ve fázi EX instrukce SL0. Tato instrukce nastavuje opět registr carry na 0 (signál `ALU_Cout = 0`). Ve 4. taktu postoupí do fáze EX instrukce SL1, která nastavuje registr carry na hodnotu 1 (`ALU_Cout = 1`). Programový čítač vystaví následující adresu 04h, ze které je načtena instrukce 7080. V tomto taktu je však také dekódována instrukce `JNC 06h` (❶). V registru carry je však stále ještě stará hodnota a to 0. Dekodér proto musí zastavit chod programového čítače a zakázat zápis do instrukčního registru (signály `PC_EN`

a `WE_IR`) a čekat na další takt. Všimněme si nastavení řídicích signálu `increase` a `JmpRetCall` (❷). Podle tohoto nastavení by měl být skok proveden. Teprve v pátem taktu dostává dekodér korektní hodnotu z registru carry a zjišťuje, že skok nebude proveden. Adresa vystavená programovým čítačem v předchozím taktu byla tedy správná. Načtená instrukce z této adresy se nemusí zahazovat. Dekodér proto pouze aktualizuje řídicí signály pro PC (`increase = 1, JmpRetCall = 0, PC_EN = 1`) a povolí zápis do IR (`WE_IR = 1`). V 6. taktu tedy mohla postoupit instrukce načtená ve 4. taktu do fáze ID (❸). Provedení podmíněného skoku v případě, že se neuskutečnil, trvalo 2 takty.

V 8. taktu instrukce `SL0 ra` opět posouvá obsah registru ra doleva a opět tak nastavuje obsah registru carry na hodnotu 1 (původně měl hodnotu 0) (❹). Programový čítač normálně inkrementuje svou hodnotu na 07h a z této adresy je načtena instrukce (0000). V tomto taktu ale opět do fáze ID přichází instrukce JC. Instrukce opět musí čekat na vstup z registru carry. Ten přichází až v 9. taktu a má hodnotu 1. Skok tedy bude proveden na adresu 00h. Dekodér tedy musí povolit činnost programového čítače (`PC_EN = 1`) (❺). Zápis do instrukčního registru je však stále zakázán (❻). V 10. taktu vystaví PC korektní adresu (00h), je načtena instrukce z této adresy (7040) (❼). Dekodér může nastavit řídicí signály na standardní hodnoty. Provedení podmíněného skoku v případě, že se uskutečnil, trvalo 3 takty.

5.5.5 Výsledky syntézy

Všechny tři verze skalárního procesoru bylo možné syntetizovat. Tato kapitola představuje výsledky syntézy verze 1.1 protože dosáhla nejlepších výsledků. Syntéza proběhla do FPGA architektury Spartan 3 velikosti XC3S200. Jako implementační jazyk byl zvolen VHDL, jako simulátor ModelSim PE.

Procesor může pracovat na frekvenci **82,923 MHz**.

Tabulka 10 zobrazuje výsledný report syntézy týkající se spotřeby funkčních bloků.
V tabulce 11 pak můžeme vidět skutečné výsledky získané z implementace designu.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	315	1920	16%
Number of Slice Flip Flops	213	3840	5%
Number of 4 input LUTs	808	3840	21%
Number of bonded IOBs	28	141	19%
Number of BRAMs	2	12	16%
Number of GCLKs	1	8	12%

Tabulka 10 – Odhad spotřeby funkčních bloků

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	213	3,840	5%	
Number used as Flip Flops	82			
Number used as Latches	131			
Number of 4 input LUTs	809	3,840	21%	
Number of occupied Slices	489	1,920	25%	
Number of Slices containing only related logic	489	489	100%	
Number of Slices containing unrelated logic	0	489	0%	
Total Number of 4 input LUTs	809	3,840	21%	
Number used as logic	553			
Number used for Dual Port RAMs	256			
Number of bonded IOBs	28	141	19%	
IOB Flip Flops	2			
Number of RAMB16s	2	12	16%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	4.58			

Tabulka 11 – Skutečná spotřeba funkčních bloků

5.6 Srovnání subskalární a skalárních architektur.

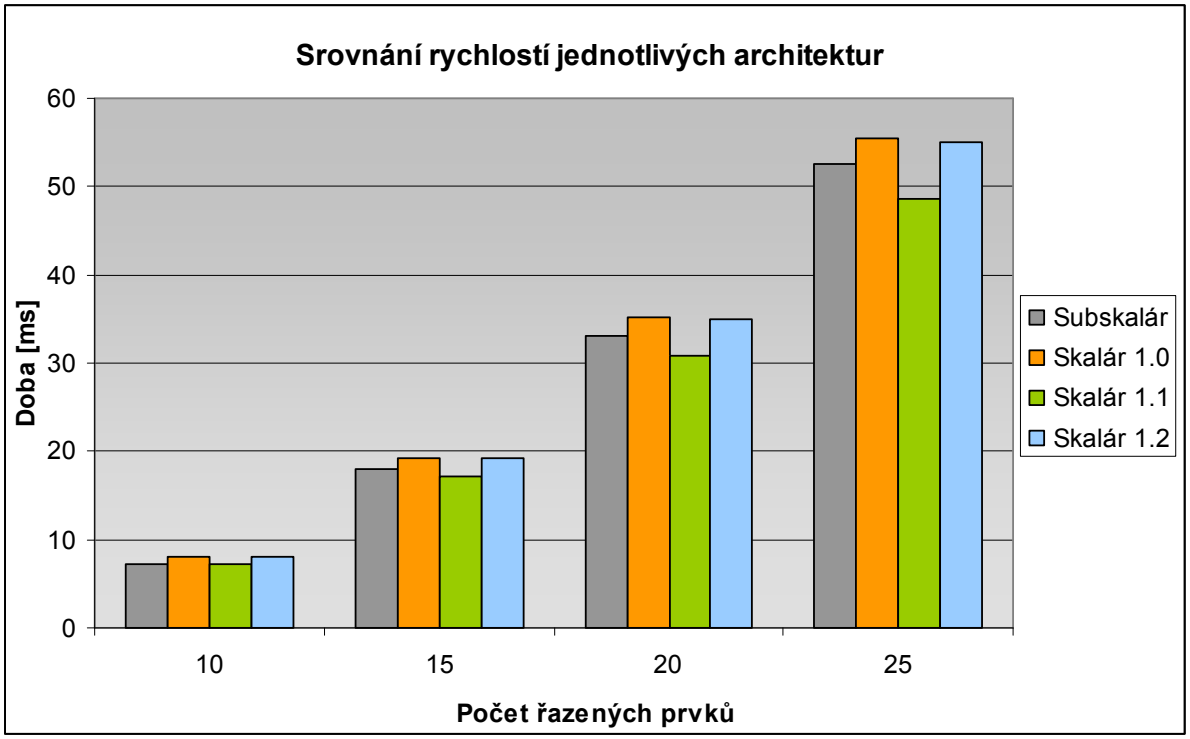
Tato kapitola pojednává o srovnání výkonosti skalární a subskalární architektury. Srovnávány jsou architektury v nichž není implementována logika cache. Pro porovnání jsem použil algoritmus bubble sort. Postupně jsem zvětšoval délku řazeného pole od 10ti do 25ti prvků. Bohužel se mi nepovedlo seřadit větší množství prvků. Od určité doby běhu algoritmu začal dávat simulátor nekorektní výsledky. Protože všechny ostatní vstupní posloupnosti menších rozsahů byly seřazeny korekntě vyloučil bych chybu algoritmu i chybu v logice procesoru. je možné, že studentská verze simulátoru ModelSim PE Student Edition 6.6a má omezení na délku běhu simulace nebo se jedná o chybu simulátoru. Pro více reprezentativní výsledky by bylo potřeba vytvořit větší skupinu algoritmů, které poběží alespoň jednotky sekund.

Tabulka 12 zobrazuje parametry srovnávaných architektur. Bylo provedeno strovnání subskalární architektury se třemi verzemi skalární architektury.

Architektura	Frekvence [MHz]	Pokuta u nepodmíněných skoků	Pokuta u podmíněných skoků
Subskalární	58,3	0	0
Skalární v1.0	82,9	1	2
Skalární v1.1	82,9	1	1-2
Skalární v1.2	61,9	1	0-1

Tabulka 12 – Parametry srovnávaných architektur

Graf na obrázku 38 znázorňuje za jak dlouho jednotlivé architektury seřadili vstupní posloupnost určité délky. Vstupní posloupnost byla pro všechny architektury stejná.



Obr 38 – Srovnání výkonnosti jednotlivých architektur

Z grafu vyplývá, že nejrychlejší byla vždy skalární architektura s optimalizovaným řešením řídicích konfliktů. Ostatní řešení dávají dokonce horší výsledky než subskalární architektura. To přikládám hlavně tomu, že zdrojový kód algoritmu byl poměrně krátký a obsahoval hodně skoků. To se negativně odráží na jakékoli zřetězené architektuře.

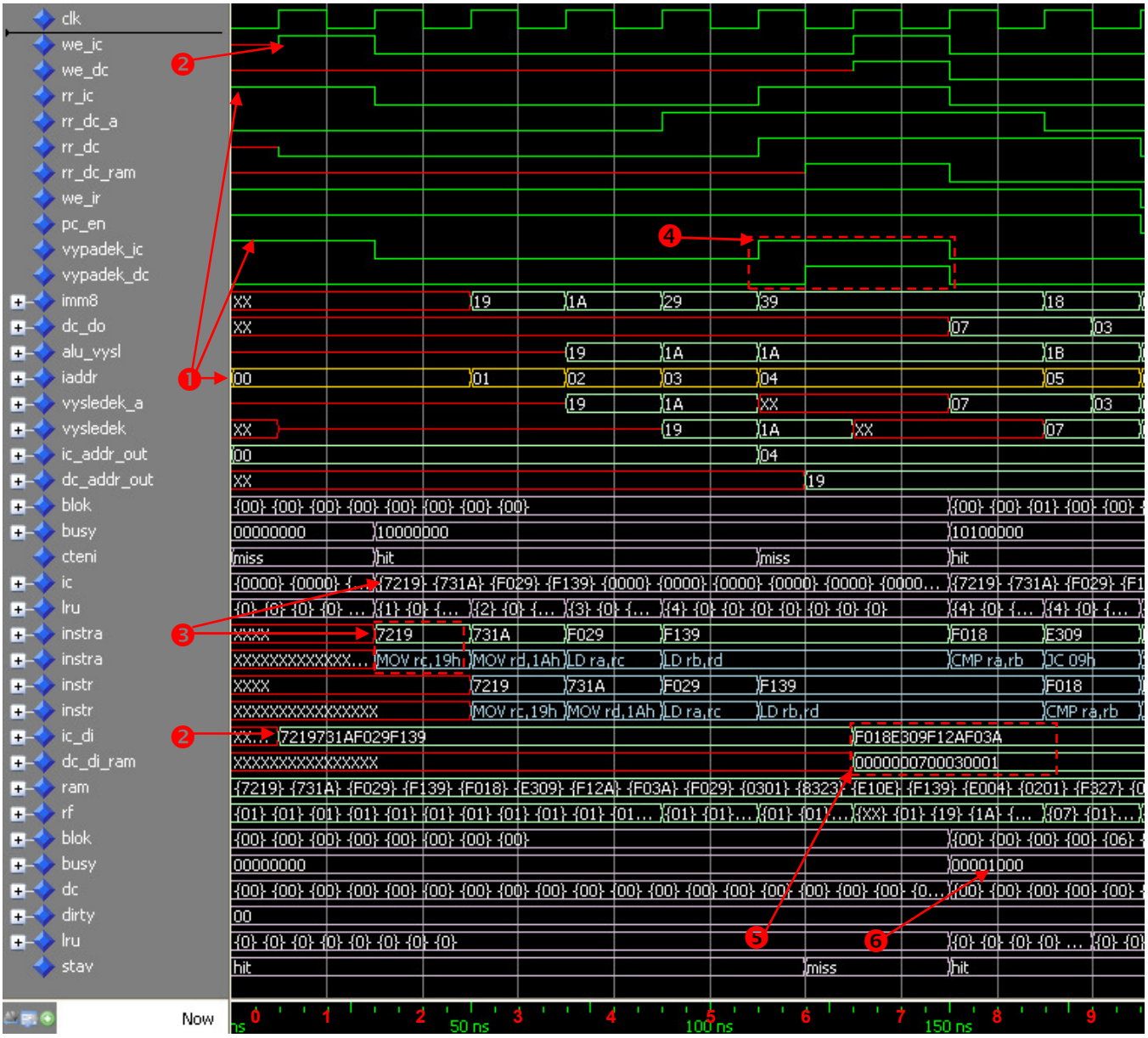
Subskalární architektura už nenabízí žádné možnosti zlepšení výkonnosti. Skalární architektury by bylo možné vylepšit zkrácením kritické cesty. Bylo by možné upravit HW předávání dat tak aby proběhlo až ve fázi EX. Tím by se značně odlehčilo fázi ID (připomínám, že ve fázi ID je instrukce dekódována a jsou načítány zdrojové operandy, které jsou dále multiplexovány s výsledky z fáze EX). Bylo by možné zavést pozitivní predikci skoků a snížit tak pokuty u nepodmíněných skoků.

5.7 Návrh skalární architektury s cache

Tato architektura vychází ze dříve implementované skalární architektury architektury verze 1.2. Byly provedeny stejné změny, které bylo nutné udělat při přechodu od subskalární verze bez pamětí cache k verzi s pamětmi cache. Dále bylo potřeba přivést do registrů IR a DR signály z DC zajazující zápis do těchto registrů v případě výpadků z DC. Pokud by totiž nastal výpadek z DC a zastavil by se pouze chod programového čítače, dobíhaly by ještě instrukce, které byly v okamžiku výpadku ve fázích IF a ID. Podrobné schéma této architektury je umístěno v přílohách.

5.7.1 Příklady

Činnost procesoru je kombinací zpracování výpadků, tak jak bylo demonstrováno u subskalární architektury s cache (kapitola 5.3), a proudového zpracování instrukcí, tak jak bylo demonstrováno u skalární architektury (kapitola 5.5). Demonstraci činnosti proto provedu na příkladu zpracování algoritmu Bubble sort. Demonstrace tohoto algoritmu je vidět na obrázku 39.



Obr 39 – Wave diagram znázorňující výpadky z IC a DC u skalární architektury

Na začátku je vystavena nultá adresa do instrukční cache. IC v tuto chvíli nemá načten žádný blok instrukcí a proto nastává výpadek z IC (❶). Výpadek je obsluhován stejným způsobem jako u subskalární architektury. S nástupnou hranou CLK obdrží paměť RAM požadavek na čtení (signál RR_IC), vystaví proto blok instrukcí a nastaví potvrzovací signál WE_IC (❷). S následující nástupnou hranou CLK jsou tyto data zapsána do instrukční cache a může být načtena první instrukce (❸). První dvě instrukce jsou typu MOV, třetí instrukce je typu LD. Ve 4. taktu je načítána instrukce LD ra,rc. V 5. taktu je dekódována. V 6. taktu přichází do fáze EX. Protože je datová cache ještě prázdná, nastává výpadek z DC (výpadky z DC nastávají ze synchronizačních důvodů se sestupnou hranou CLK). S nástupnou hranou 6. taktu však zároveň programový čítač vystavuje adresu 04h. Instrukce z této adresy však již není v IC a proto nastává výpadek z IC. Současně tedy nastaly výpadky z IC i DC a současně jsou i obsluhovány (❹). Z diagramu je vidět, jak paměť RAM vystaví v 7. taktu oba načtené bloky dat (❺). Blok dat pro DC začíná hodnotou

0000 protože sekce s daty je uložena v paměti RAM od adresy 25dec, což je 6. blok v RAM, druhá pozice v bloku (počítáno od 1). Tento blok je mapován na 3. set v DC a první volný blok v tomto setu. 6. blok z RAM je tedy v tomto případě mapován na 5. blok v DC. Umístění bloku na tuto pozici je možné rozpoznat podle nastavení příslušného busy bitu (❻). V 8. taktu jsou vstupní bloky z RAM uloženy do IC a DC a může pokračovat zpracování instrukcí z dalšího bloku.

Obě cache jsou blokující ve smyslu, že v okamžiku výpadku je zastaven chod programového čítače. Nicméně nic nebrání tomu aby v případě výpadku z IC pokračovaly instrukce, které se nachází ve fázích ID a EX, stupni procesoru a mohly být dokončeny. V případě výpadku z DC toto však není možné a musí být zastaven chod celého procesoru.

5.8 Superskalární architektura

Superskalární architekturu používají současné procesory. Tato kapitola je zde zařazena pouze pro podrobnější porovnání současné architektury s architekturami implementovanými v rámci diplomové práci.

Rozdíl oproti skalární architektuře spočívá v tom, že instrukce mohou být prováděny skutečně paralelně, tedy stupně EX se mohou překrývat. Je to dáno replikací určitých částí procesoru (zejména funkčních jednotek) a rozdělením instrukcí do skupin. Každá skupina je pak prováděna nezávisle na ostatních. Procesor může mít tak např. dvě L/S jednotky, jednu ALU, jednu FP jednotku atd. Tomuto paralelismu se říká prostorový paralelismus. Jedná se o opravdový paralelismus kdy jsou instrukce prováděny skutečně paralelně. Stejně jako u skalární architektury platí i zde, aby mohly být instrukce prováděny paralelně, je potřeba co nejvíce na sobě nezávislých instrukcí.

5.8.1 Mezioperační latence

S nárůstem frekvence a zvyšující se komplexností instrukcí už není pravidlem, že by průchod instrukce jednou částí procesoru trval pouze jeden takt. Tyto části jsou dále děleny a proto je běžné, že komplexní instrukce může strávit ve funkční jednotce několik taktů. V následujícím příkladě instrukce `fadd` stráví ve fázi EX celkem tři takty. Závislá instrukce `sd` musí dva takty čekat, tak jak je znázorněno v tabulce 13 než dostane výsledek předchozí instrukce. Této době se říká mezioperační latence.

fadd	IF	ID	EX	EX	EX	WB	
sd		IF	ID	-	-	EX	WB

Tabulka 13 – Mezioperační latence 1

Uvažujme následující příklad

```
zacatek: fld f0,[r0]
         ------
         fadd f4,f0,f2
         ------
         ------
         sd [r1],f4
         subi r1,r1,#8
         bnez r1,zacatek
         ------
```

9 taktů

Mezioperační latence v uvedeném příkladě jsou následující

FLD,BNEZ…1 takt
FADD…2 takty

Zpracování uvedených instrukcí je znázorněno v tabulce 14.

Takt	1	2	3	4	5	6	7	8	9	10	11	12
fld	IF	ID	EX	EX	WB							
fadd		IF	ID	-	EX	EX	EX	WB				
sd			IF	ID	-	-	-	EX	WB			
subi				IF	-	-	-	ID	EX	WB		
bnez					IF	-	-	-	ID	EX	EX	WB

Tabulka 14 – Mezioperační latence 2

Pokud bychom neuvažovali naplnění linky a konečný „zápis“ výsledku skoku tak bude průběh jedné smyčky trvat celkem 9 taktů. Otázkou je, jak tyto „prázdná místa“ zaplnit?

Kód smyčky je možné pak přepsat takto:

```
zacatek: fld f0,[r0]
         ----------
         fadd f4,f0,f2
         subi r1,r1,#8
         bnez r1,zacatek
         sd [r1+8],f4
```

Přesunutím instrukcí `subi` a `bnez` se nám podařilo zaplnit mezeru za instrukcí `fadd` a mezeru za `bnez`. Přesunutí možná vypadá na první pohled nelogicky, ale má své opodstatnění. Instrukce `subi` dekrementuje obsah registru r1 o hodnotu 8, proto musím tutéž hodnotu přičíst v poslední instrukci `sd` aby byl výsledek uložen na správné místo. Instrukce `bnez` pak skočí na dané návěští. Na první pohled to vypadá, že bude skok proveden ještě před uložením výsledku a k uložení výsledku tak nikdy nedojde. Ve skutečnosti provedení instrukce `bnez` trvá dva takty a proto se stihne výsledek uložit korektně do paměti.

Přeskládávání instrukcí je typické pro superskalární architekturu. Provádí ho kompilátor nebo procesor přímo za běhu programu. Samozřejmě aby toto mohlo být uskutečněno musí být známy mezioperační latence pro jednotlivé intrukce.

5.8.2 Popis částí procesoru

Instrukční cache

IC se nijak neliší od IC u skalární architektury. Rozdíl je ale v načítání instrukcí. Nenačítá se pouze jedna aktuální instrukce, na kterou ukazuje PC, ale načítá se celý blok instrukcí (v našem případě 4).

Instrukční dekodér

Paralelně dekóduje všechny načtené instrukce a předává je jednotce DI (Dispatch).

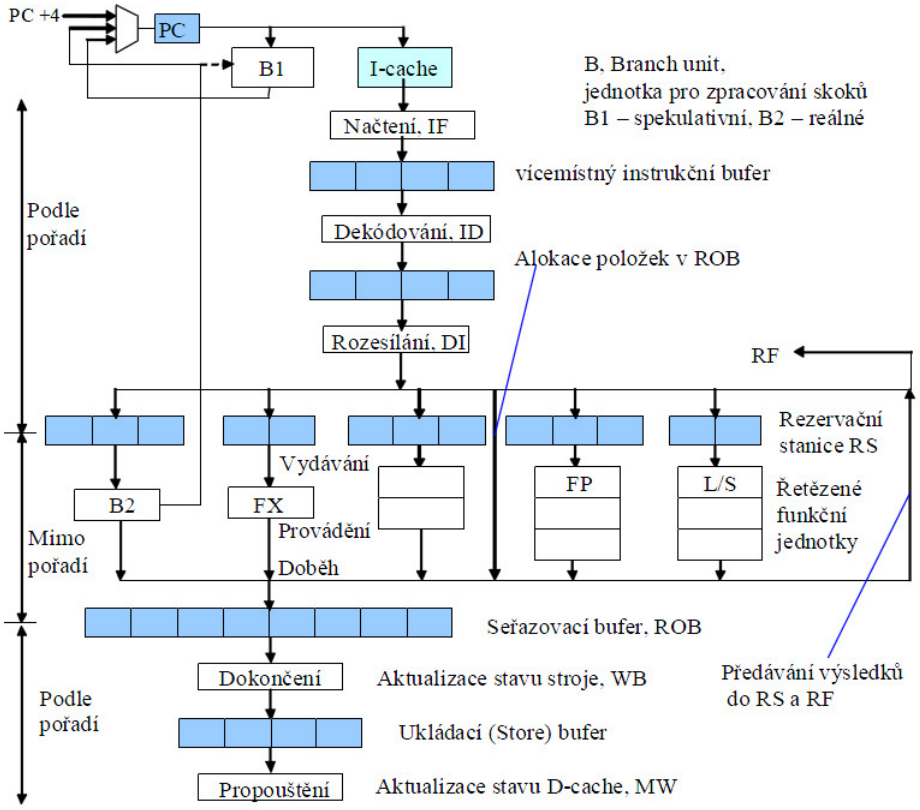
Jednotka pro rozesílání instrukcí

Jednotka DI rozesílá dekódované instrukce do reservačních stanic jednotlivých funkčních jednotek. Reservační stanici si na začátku můžeme představit jako jakýsi buffer, který zásobuje jednotlivé funkční jednotky instrukcemi. Od této chvíle jsou instrukce prováděny paralelně *mimo své původní pořadí*. Instrukce jsou tedy zpracovány ve svém původním pořadí pouze do okamžiku kdy jsou rozeslány do jednotlivých reservačních stanic (viz obrázek výše).

Instrukce odesílaná do nějaké reservační stanice je zároveň odesílána do tzv. seřaďovacího bufferu (Reorder Buffer - ROB). V tomto bufferu jsou tedy instrukce uloženy v původním programovém pořadí!!!

Instrukce jsou ve funkčních jednotkách tedy prováděny paralelně mimo své původní pořadí. V okamžiku kdy je instrukce provedena (v rámci funkční jednotky). Je výsledek okamžitě poslán po sběrnici (**Common Data Bus** - CDB) a pokud ho potřebuje některá z následujících instrukcí tak je zapsán do příslušné reservační stanice. Toto ještě automaticky neznamená, že bude výsledek zapsán do cílového registru!!! Do cílového registru je zapsán vždy jenom výsledek té instrukce, která je na čele ROB. Důvody budou uvedeny v dalších kapitolách. Kompletní schéma superskalární architektury je znázorněno na obrázku 40.

Schéma superskalární architektury



Obr. 40 Blokové schéma superskalární architektury, převzato z [9]

Reservační stanice s hodnotami operandů a eliminace konfliktů

Busy bit	fce	op. 1	V1	op. 2	V2	dst	rTag	Ready bit

Tabulka 15 Položka reservační stanice

Busy bit – udává jestli je položka reservační stanice obsazená nebo ne
fce – blíže popisuje instrukci. Např. pokud se jedná o RS pro L/S jednotku tak zde bude uvedeno jestli se jedná o operaci load nebo store
op. 1 – operand 1
V1 – bit platnosti operandu 1 (V…valid)
op. 2 – operand 2
V2 – bit platnosti operandu 2 (V…valid)
dst – adresa cílového registru
rTag – příznak výsledku (Result Tag)
Ready bit – udává jestli jsou oba operandy platné

V	tag	operand

Tabulka 16 Položka registrového pole

V – bit platnosti
tag – příznak
operand – samotný operand

Popis činnosti a eliminace konfliktů

1) Pokud je operand v RF platný, má nastavený V = 1. Na začátku se v RS vynulují bity platnosti pro oba operandy. Pokud jsou oba zdrojové operandy platné (a to pro první instrukci určitě jsou), jsou načteny do RS včetně bitů platnosti. Pokud některý ze zdrojových operandů není platný, je na pozici daného operandu v RS uložen příznak (políčko tag v RF) zdrojového operandu z RF a dále je samozřejmě načten bit platnosti V = 0.

2) Pro cílový operand je v RF vynulován bit platnosti. Dále je pro cílový operand vygenerován příznak, který je uložen jak do položky `rTag` v RS tak do položky `tag` v RF. Pokud je za sebou více instrukcí, které mají stejný cílový registr, tak je tento příznak přepsán. **Tedy pokud jde po sobě několik instrukcí se stejným cílovým registrem, tak pouze ta poslední má příznak výsledku jak v RS tak v RF!!!**

3) Pokud jsou oba zdrojové operandy platné, je nastaven Ready bit = 1 a instrukce je z RS poslána do funkční jednotky (pokud je volná). Takto je eliminován konflikt **RAW** – instrukce čeká v reservační stanici dokud nemá platné oba zdrojové operandy.

4) Instrukce je provedena a výsledek je zaslán spolu s adresou cílového registru a příznakem výsledku po sběrnici CDB. Pokud některá z instrukcí čeká v RS na výsledek této instrukce (což se pozná podle shody příznaků) tak je tento výsledek zapsán přímo do reservační stanice, bit platnosti tohoto operandu je nastaven na 1.

Výsledek není automaticky zapsán do cílového registru RF. Pokud se v cílovém registru vyskytuje jiný příznak než příznak současného výsledku (což se může stát tím, že mělo několik zpracovávaných instrukcí stejný cílový registr a postupně si příznak přepisovaly), tak se výsledek do cílového registru vůbec nezapíše!!! Tedy pokud je za sebou několik instrukcí se stejným cílovým registrem a tyto instrukce jsou načteny jako jeden blok, tak pouze ta poslední zapíše svůj výsledek do RF. Takto jsou eliminovány konflikty **WAW** a **WAR**.

Uvažujme následující příklad:

```
mul r1,r5,r1
add r6,r1,r3
add r1,r2,r3
```

Instrukce mul trvá o hodně déle než obě instrukce add.

RAW – druhá instrukce potřebuje ke svému výpočtu výsledek první instrukce – `add` bude čekat v RS protože instrukce `mul` nastavila bit platnosti registru r1 na nulu. Instrukce `mul` dále vygenerovala příznak výsledku např. „x001“ a uložila ho jak do RS tak do RF. Instrukce `add` nemůže načíst operand r1, načte tedy místo něj na pozici operandu 1 pouze příznak x001.

Třetí instrukce je nezávislá na prvních dvouch. Oba operandy má platné. Vynuluje bit platnosti registru r1 (ten je ve skutečnosti už vynulován instrukcí mul) a vygeneruje příznak výsledku např. „x003“. Příznak opět uloží do RS i RF, čímž, ale přepíše v RF příznak uložený instrukcí `mul`. Třetí instrukce je dokončena a kontroluje se jestli nějaká instrukce v RS nečeká na operand r1. Na ten sice čeká druhá instrukce, ale ta čeká na výsledek

s příznakem x001. Výsledek třetí instrukce proto není zapsán do RS, ale je zapsán do RF (příznaky v RF se shodují). Takto je eliminován **WAR**.

Instrukce `mul` je provedena a posílá svůj výsledek po CDB. Opět se kontroluje jestli v RS nečeká nějaká instrukce na výsledek v registru r1. Na ten čeká druhá instrukce, jsou zkontrolovány příznaky a ty se shodují. Výsledek je proto zapsán do RS. Dále by měl být zapsán do samotného architekturního registru r1. Tam se ale neshodují příznaky proto tam zapsán není. Takto je eliminován **WAW**.

Reservační stanice bez hodnot operandů a eliminace konfliktů

Struktura položky reservační stanice:

Busy bit	fce	Adresa op. 1	V1	Adresa op. 2	V2	dst	Ready bit

Tabulka 17 - Položka RS bez hodnot operandů

V	operand

Tabulka 18 – Položka registrového pole v případě RS bez hodnot operandů

Oproti rezervačním stanicím s operandy jsou zde dva rozdíly. Položka RS neobsahuje operand samotný, ale pouze jeho adresu do RF. Nejsou zde obsaženy příznaky výsledku – ani v RS ani v RF.

Popis činnosti a eliminace konfliktů.

1) Zdrojové registry v RF a RS obsahují **na začátku** vždy shodné bity platnosti.

2) Instrukce si reservuje cílový registr tak, že vynuluje bit platnosti v RF. Pokud nějaká další instrukce má stejný cílový registr, bude muset čekat než bude v RF nastaven bit platnosti předchozí instrukcí na 1. Takto je eliminován konflikt **WAW**.

3) Pokud jsou oba operandy platné a je volná FJ, tak je instrukce poslána do FJ a zároveň jsou z RF do FJ načteny oba zdrojové operandy. Takto je eliminován konflikt **RAW**.

4) Instrukce je provedena. Výsledek je zapsán do RF pouze pokud v RS není nějaká instrukce se stejnou adresou zdrojového operandu a bitem platnosti nastaveným na 1. Takto je eliminován konflikt **WAR**.

Příklady:

```
mul r1,r2,r3 }
add r5,r1,r2 } RAW
```

Instrukce `mul` nastaví v RF bit platnosti pro r1 na nulu. Instrukce `add` nemůže být provedena protože nemá platný operand r1 v RF. Teprve po zápisu výsledku mul do RF se začne provádět instrukce add.

```
mul r1,r2,r3 }
add r1,r2,r3 } WAW
```

Instrukce `mul` opět nastaví bit platnosti v RF operandu r1 na nulu. Instrukce `add` nemůže být provedena protože už má vynulovaný bit platnosti pro cílový operand. Musí tedy čekat na dokončení předchozí instrukce

```
mul r1,r2,r3
add r3,r2,r1 }WAW
sub r2,r5,r5 }
```

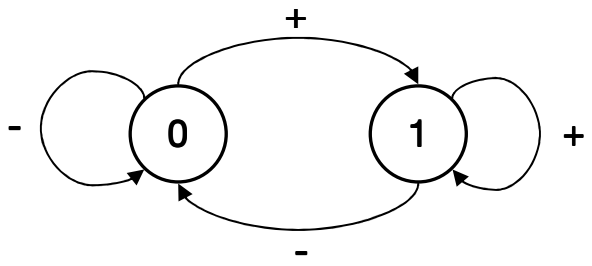
Paralelně můžou probíhat instrukce `sub` a `mul`. Instrukce `sub` však nemůže svůj výsledek zapsat do r2 dokud nenačte operand r2 instrukce `add`. (Instrukce `add` má nastaven bit platnosti pro r2 v RS na 1.)

Prediktory skoků

Další novinkou superskalárních procesorů jsou prediktory skoků. Predikuje se podmínka (jestli se skočí) a/nebo cílová adresa (kam se skočí) [9].

Predikce podmínky

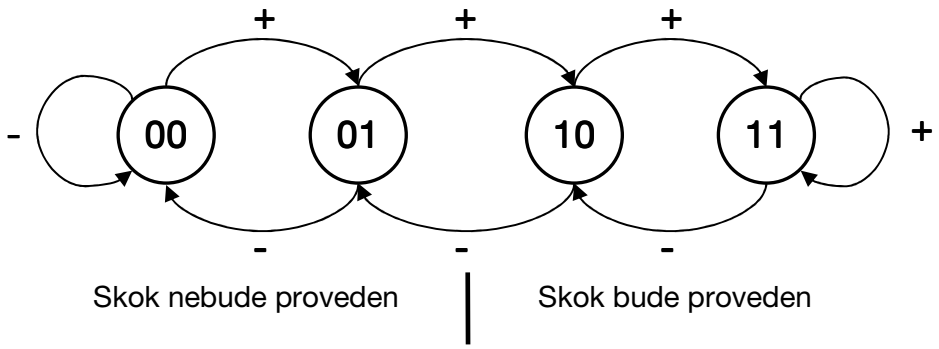
1b prediktor



Obr. 41 -1b prediktor

Na začátku si zvolíme např. negativní predikci, tedy, že skok nebude proveden (stav 0). Pokud skok bude proveden, přechází se do stavu 1 a uvažuje se pozitivní predikce. Jinými slovy, pokud byl předchozí skok proveden, prediktor počítá s tím, že bude proveden i tento, pokud nebyl předchozí skok proveden, nebude ani tento. Stavový diagram znázorňuje obr. 41.

2b prediktor



Obr. 42- 2b prediktor

2. prediktor bere v úvahu delší historii skoků. Pokud tedy vezmeme na začátku negativní predikci (stav 00), tak pro přepnutí do pozitivní predikce musí být za sebou dvakrát provedeny skoky tak jak je znázorněno ve stavovém diagramu na obr 42.

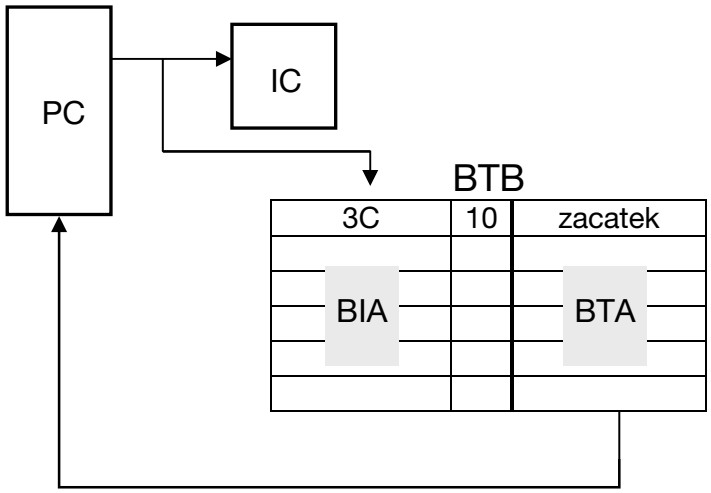
Tyto prediktory jsou implementovány pomocí tabulky BHT (Branch History Table). Tato tabulka by pro 2b prediktor a níže zobrazený kód mohla vypadat následovně:

Adresa skoku	Stav
.	.
.	.
3C	10
.	.
.	.

zacatek:

Adresa	Instrukce
3A	add …
3B	sub …
3C	jnz zacatek

Predikce podmínky i cílové adresy



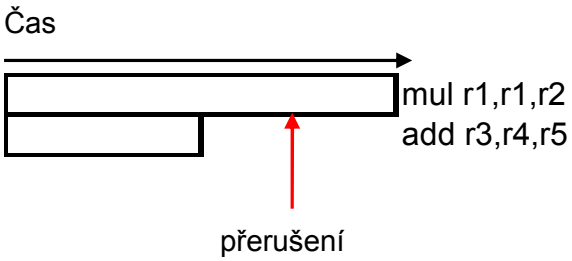
Obr. 43- Prediktor podmínky i cílové adresy

Zkratka BTB znamená Branch Target Buffer. Tato tabulka by se dala rozdělit na dvě podtabulky. BIA – Branch Instruction Address je pouze jiný název pro BHT. BTA – Branch Target Address je tabulka obsahující adresu kam se bude skákat.

Programový čítač tedy posílá adresu následující instrukce do instrukční cache a zároveň do jednotky pro predikci skoků. Pokud se v jednotce pro predikci skoku najde aktuální adresa, znamená to, že tato instrukce je instrukce skoku. Podle predikce se rozhodne zda bude skok proveden či nikoli. Pokud bude proveden načte se do PC cílová adresa skoku. Pokud se záznam v BTB nenajde a přesto se zjistí, že šlo o skok tak je záznam do tabulky vložen. Zapojení ilustruje schéma na obr. 43.

Reorder Buffer (ROB) - nepřesné přerušení

Problém bude uveden na následujícím příkladě



Instrukce `mul` a `add` jsou na sobě nezávislé. Instrukce `add` proběhne rychleji a nic nebrání tomu aby výsledek své operace okamžitě zapsala do registru r3. Pokud by nastalo přerušení v momentě kam ukazuje šipka, musel by být vyvolán obslužný kód přerušení a

ten by pracoval se špatným výsledkem v registru r3. Výsledek se do architekturních registrů tedy nesmí zapisovat dříve než jsou dokončeny všechny předchozí instrukce.

Toto je zajištěno pomocí ROB. Ten obsahuje ne jenom samotnou instrukci, ale také informaci v jaké fázi se instrukce nachází (RS/FJ/DONE) nebo jestli je výsledek skoku spekulativní nebo potvrzený. Výsledek instrukce je zapsán do architekturního registru pouze pokud je instrukce na čele ROB.

Instrukce jsou do ROB vydávány jednotkou DI pro rozesílání v programovém pořadí.

5.8.3 Shrnutí

Superskalární procesory přináší skutečný prostorový paralelismus. Instrukce jsou prováděny paralelně mimo své původní pořadí. Kromě jiných komponent je zavedena predikce výsledku skoku a spekulativní provádění těchto skoků. Rysem této architektury je, že jsou instrukce prováděny mimo své programové pořadí [6], [7], [9], [11].

6 Závěr

Úkolem této práce bylo prostudovat zřetězené architektury procesorů a vyrovnávací paměti cache. Tuto architekturu navrhnout včetně logiky pamětí cache a implementovat v jazyce VHDL. Implementované architektury měly být odsimulovány v simulačním prostředí ModelSim. Závěrem se pokusit implementovanou architekturu syntetizovat do FPGA a porovnat s nezřetězenou architekturou. Samotný návrh nebyl složitý, při postupné implementaci se však ukazovaly jeho nedokonalosti a musel být mnohokrát upravován. Pro srovnání jsem provedl implementaci dvou verzí nezřetězené architektury – s logikou cache a bez ní, dále pak implementaci čtyř verzí skalární architektury – opět jedna verze s logikou cache, ostatní bez ní. Verze procesorů bez logiky cache byly syntetizovány a jejich výkonnost srovnána na algoritmu Bubble sort. Paměti cache se mi bohužel nepodařilo syntetizovat, a proto verze procesorů využívající tyto komponenty nemohly být zahrnuty do srovnání s ostatními.

Přínos této práce je spíše osobní. Implementované architektury jsem sice znal již z dřívějšího studia, ale pouze ve formě hrubého návrhu na úrovni větších komponent. Detailní návrh a implementace mi tak přinesly mnoho nových poznatků. Z hlediska přínosu pro vědu a výzkum není tato práce ničím novým. Vývojem procesorů se zabývají mnoho lidí řadu let, a proto jsem neočekával, že by práce mohla přinést objev nebo vylepšení nějaké ze současných architektur.

Implementovanou skalární architekturu je možné dále rozšiřovat a snažit se dosáhnout vyšší výkonnosti. První krok by měl začít studiem kritické cesty a snaha o její zkrácení. Pro praktickou použitelnost by bylo dobré změnit instrukční sadu z load/store na jinou, podporující více adresovacích režimů. To by pravděpodobně znamenalo nutnost zavedení dalšího stupně – Cache access – umístěného před fází Execute. Dále by bylo možné provést ruční implementaci knihovních funkcí, např. implementaci sčítání nebo násobení, což by poskytlo možnost zřetězení těchto jednotek. V neposlední řadě je možné upravit architektury pamětí cache tak aby je bylo možné syntetizovat.

7 Literatura

[1] Martínek T.: Materiály k předmětu PCS. Dostupné online na URL http://www.fit.vutbr.cz [2007]

[2] Glauert H. W.: VHDL Tutorial. Ddostupné na online na URL http://www.vhdl-online.de/tutorial/ [2010]

[3] Pinker J., Poupa M.: Číslicové systémy a jazyk VHDL. Praha, BEN 2006.

[4] Chang K. C.: Digital Systems Design with VHDL and Synthesis. USA, 1999.

[5] Mentor Graphics Corporation: ModelSim User's Manual. Oregon, 2010.

[6] Šilc J., Robič B., Ungerer T.: Processor Architecture From Dataflow to Superscalar and Beyond. Springer Verlag, Berlin, 1999.

[7] Johnson M.: Superscalar Microprocessor Design. PTR Prentice Hall, Inc., New York, 1991

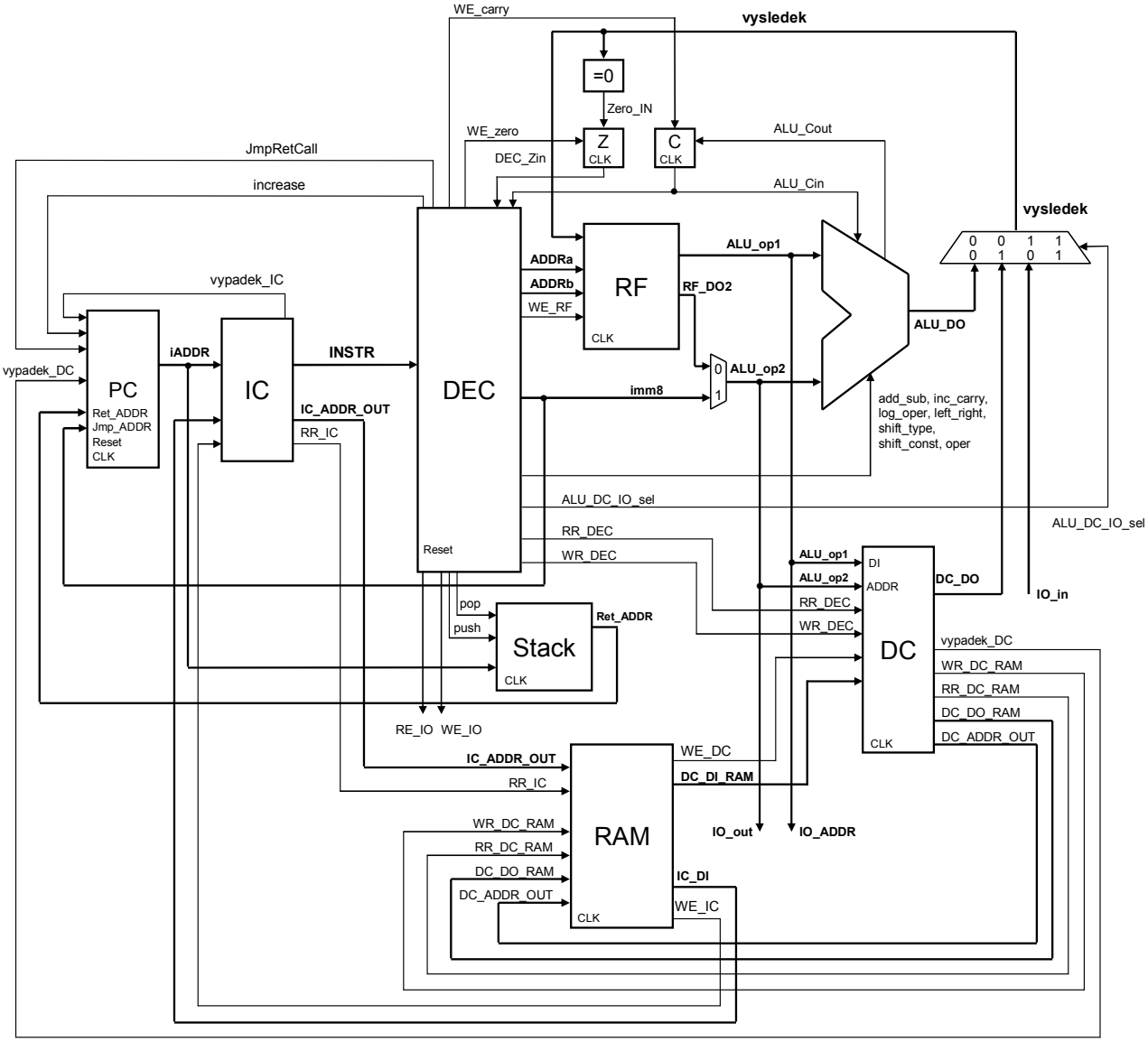
[8] Hennessy John. L., Patterson David A.: Computer Architecture A Quantitative Approach. San Fransisco, 2007.

[9] Dvořák V.: Slajdy z předmětu Architektura procesorů. Brno, 2009

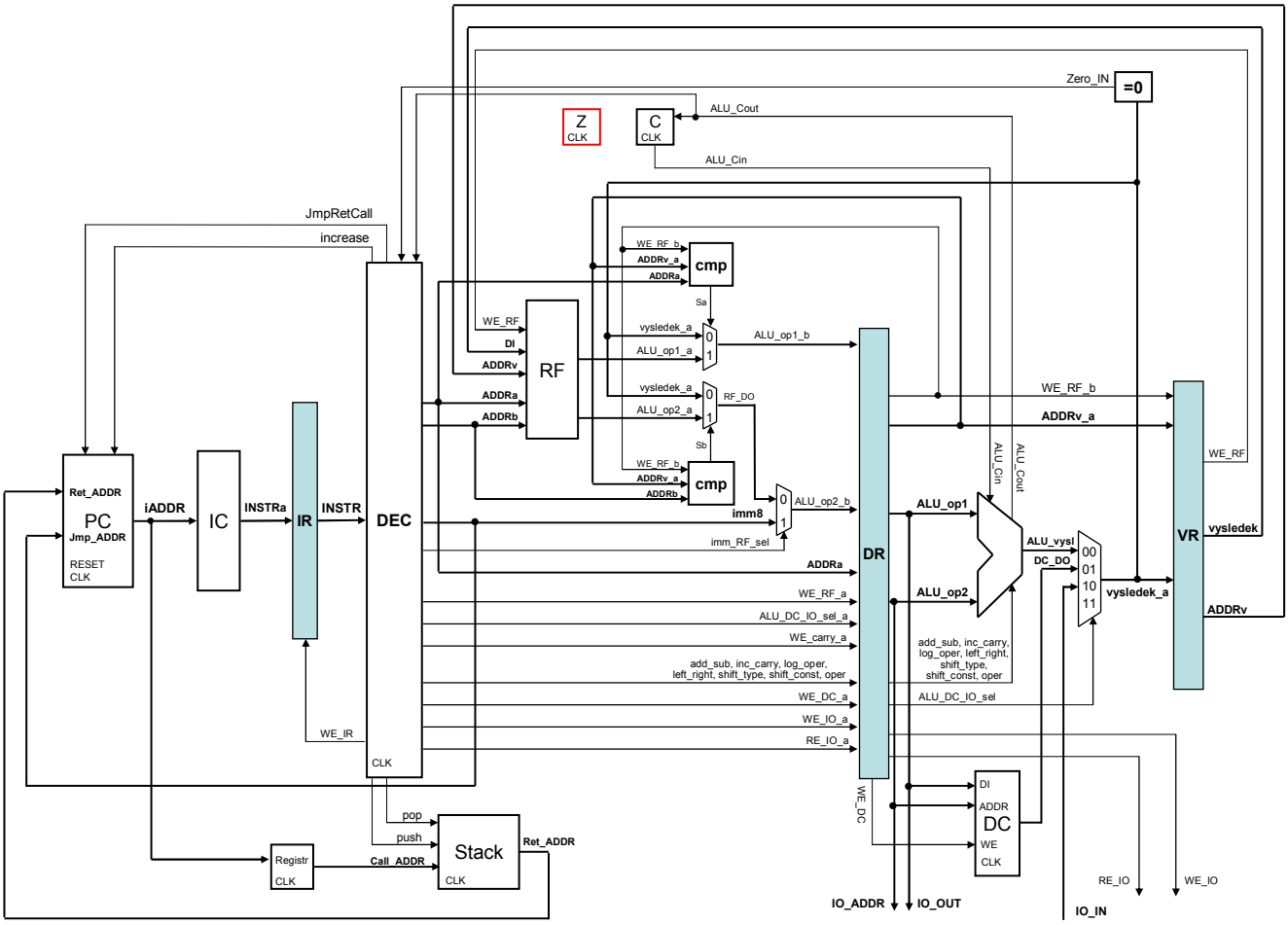
[10] Jacob B., Spencer W. Ng., Wang David T.: Memory Systems Cache, DRAM, Disk. Morgan Kaufmann Publishers, Burlington, 2008.

[11] Shen J.: Modern Processor Desing: Fundamentals of Superscalar Processors. New York, McGraw-Hill Company, 2005.

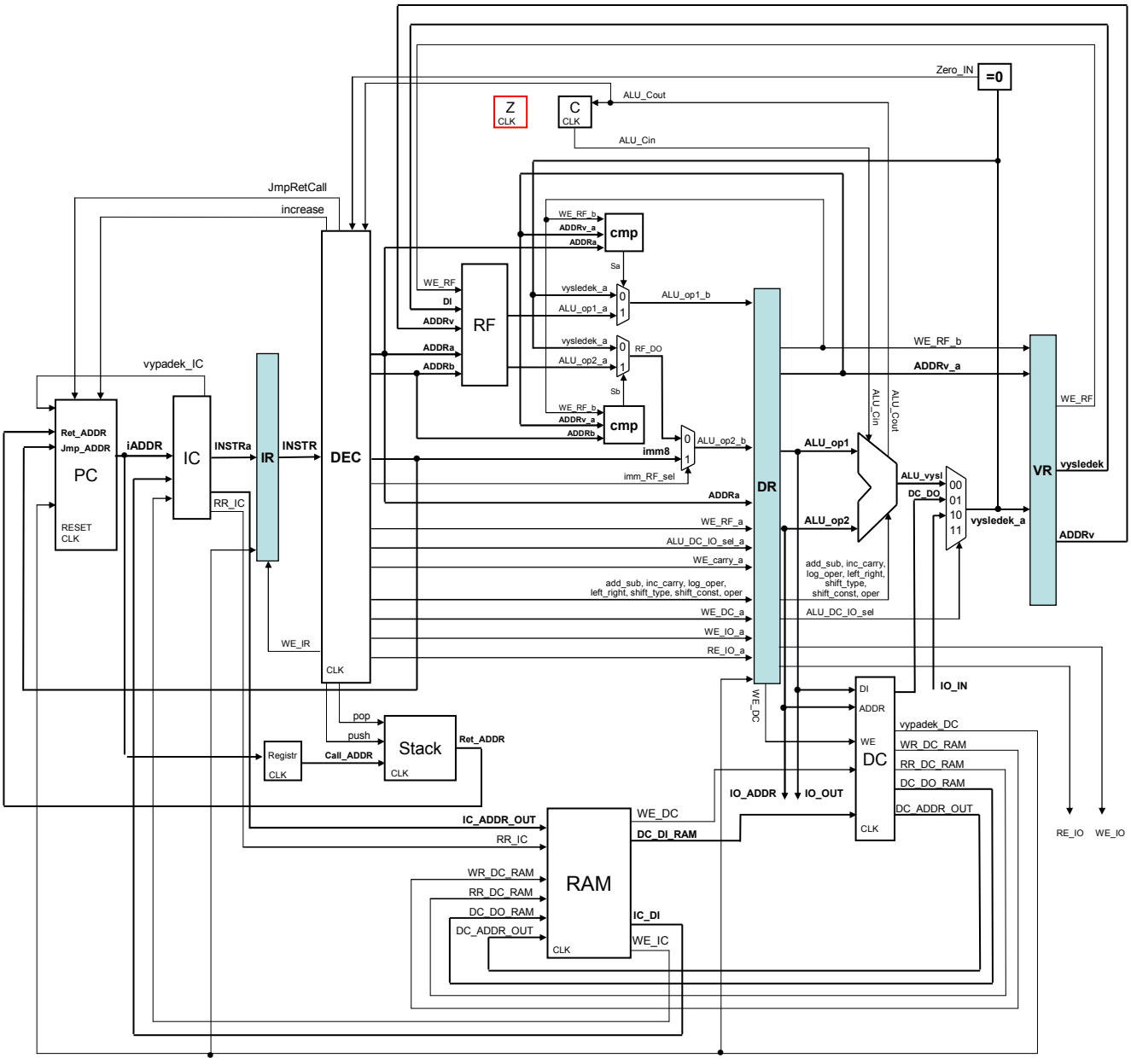
8 Přílohy



Obr. 44 – Schéma subskalární architektury s implementovanou lokikou pamětí cache



Obr. 45 – Schéma skalární architektury verze 1.2



Obr. 46 – Schéma skalární architektury verze 1.3