VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

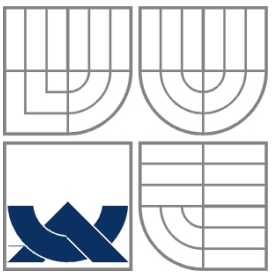
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE PROSTŘEDKŮ PRO ZVÝŠENÍ VÝKONU PROCESORU

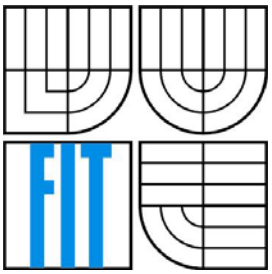
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE LUCIE ZLATOHLÁVKOVÁ
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE PROSTŘEDKŮ PRO ZVÝŠENÍ VÝKONU PROCESORU

DESIGN AND IMPLEMENTATION OF MECHANISMS FOR ENHANCING PERFORMANCE OF CPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

LUCIE ZLATOHLÁVKOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

JOSEF STRNADEL, Ing., Ph.D.

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Zlatohlávková Lucie**
Obor: Výpočetní technika a informatika
Téma: **Návrh a implementace prostředků pro zvýšení výkonu procesoru**
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s moderními prvky a principy používanými v architekturách procesorů - zejména pipelining, cache, predikce skoků. Zvažte možnosti jejich implementace ve VHDL, jejich výhody a nevýhody.
2. Po dohodě s vedoucím vyberte množinu prvků z předchozího bodu a obohaťte o ně výchozí procesor vyvinutý v rámci vašeho předchozího Ročníkového a Semestrálního projektu. Implementaci pečlivě zdokumentujte.
3. Proveďte simulaci modifikovaného procesoru a zvažte možnost jeho implementace v FPGA.
4. Vhodně zvoleným způsobem porovnejte vlastnosti modifikovaného procesoru s vlastnostmi výchozího procesoru a rovněž výsledky získané simulací modifikovaného procesoru s teoreticky očekávanými výsledky.

Literatura:

- Cohen, B. VHDL Coding Styles and Methodologies, Kluwer Academic Publishers, Dordrecht, 2001. 453 s. ISBN 0-7923-8474-1.
- Dvořák, V., Drábek , V. Architektura procesorů, VUTIUM, Brno, 1999. 293 s. ISBN 80-214-1458-8.
- Hennessy, J.L., Patterson, D.A. Computer Architecture - A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, New York, 2003. 1000 s. ISBN 1-55860-596-7.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. **Slečna**

Jméno a příjmení: **Lucie Zlatohlávková**
Id studenta: 21458
Bytem: Konradova 7, 628 00 Brno
Narozena: 22. 12. 1981, Brno
(dále jen "autor")

a

2. **Vysoké učení technické v Brně**

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Návrh a implementace prostředků pro zvýšení výkonu procesoru
Vedoucí/školitel VŠKP: Strnadel Josef, Ing., Ph.D.
Ústav: Ústav počítačových systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato diplomová práce je zaměřená na problematiku architektur procesorů. Základem projektu je návrh jednoduchého procesoru, který je obohacen o moderní prvky a principy používané v architekturách procesorů, jako jsou pipelining, cache a predikce skoků. Navržený procesor je implementován pomocí jazyka VHDL a simulován v prostředí programu ModelSim.

Klíčová slova

Architektura, Počítač, Procesor, Instrukce, Pipelining, Predikce skoků, Paměť cache.

Abstract

This master‘s thesis is focused on the issue of processor architecture. The ground of this project is a design of a simple processor, which is enriched by modern components in processor architecture such as pipelining, cache memory and branch prediction. The processor has been made in VHDL programming language and was simulated in ModelSim simulation tool.

Keywords

Architecture, Computer, Processor, Instruction, Pipelining, Branch prediction, Cache memory.

Citace

Lucie Zlatohlávková: Návrh a implementace prostředků pro zvýšení výkonu procesoru, diplomová práce, Brno, FIT VUT v Brně, 2007

Návrh a implementace prostředků pro zvýšení výkonu procesoru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením Ing. Josefa Strnadela, Ph.D.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Lucie Zlatohlávková
22. 5. 2007

Poděkování

Děkuji Ing. Josefovi Strnadelovi, Ph.D. za odborné vedení, rady a podněty, které mi během práce poskytoval.

Obsah

1 Úvod

Oblast počítačové techniky je nesmírně dynamická a již delší dobu prochází prudkým vývojem. Vyvíjejí se nové typy procesorů s vyššími taktovacími frekvencemi, nové sběrnice i operační systémy. Vylepšení souvisejí se zlepšováním výrobní technologie obvodů vysoké integrace a spočívají například ve zvyšování použitých taktovacích frekvencí, větší kapacitě paměti a použití složitějších procesorů s více výkonnými jednotkami.

Všechna tato zdokonalování spočívají v úpravách všeobecně zavedeného modelu počítače. Tento model je znám již více než 50 let a s menšími či většími obměnami z něj vychází většina dnešních počítačů. Zmíněný model předpokládá třeba to, že počítač má paměť, nad kterou aritmetická jednotka vykonává operace, že počítač je řízený programem – strojovým kódem, který se skládá z instrukcí, pracuje ve dvojkové soustavě apod.

Jak již bylo řečeno, některé základní principy týkající se architektury počítačů zůstávají stále stejné a je nutné je chápat předtím, než se začneme zabývat moderními prvky a principy, jejichž zavedením do základní architektury počítače umožníme zvýšení jeho výkonnosti.

Náplní tohoto projektu je návrh jednoduchého procesoru, který je poté implementován v jazyce VHDL. Tento procesor je dále vylepšen o některé moderní prvky a principy, které se využívají v architekturách počítačů a procesorů, jako zřetězené zpracování instrukcí, predikce skoků, paměť cache. Nakonec mělo být provedeno na základě simulací porovnání vlastností výchozího a modifikovaného procesoru a rovněž výsledků získaných simulací s teoreticky očekávanými výsledky. Tuto část práce se mi však nepodařilo realizovat z důvodu nefunkční implementace.

Ve dvou úvodních kapitolách této práce budou postupně vyloženy základní pojmy, které se týkají architektur počítačů a procesorů a vybraných moderních prvků a principů, zřetězeného zpracování instrukcí, predikce skoků a paměti cache, které budou dále používány v návrhu. Poté bude následovat kapitola týkající se návrhu jednoduchého procesoru. Bude zde zdůvodněna použitá architektura procesoru. Podrobně se budeme zabývat návrhem jednotlivých komponent procesoru a návrhem instrukční sady. V dalších třech kapitolách budeme postupně pokračovat návrhem pamětí cache, zřetězeného zpracování instrukcí a prediktoru skoků. Každá kapitola bude věnována jednomu z těchto prvků a bude zde také vysvětleno, jak daný prvek modifikuje původně navržený procesor. Následující kapitola měla být věnována porovnání vlastností jednoduchého procesoru, jehož návrhem jsme se zabývali ve čtvrté kapitole, a jeho modifikovaných verzí s přidanými prvky, které byly popsány také v předchozích kapitolách, a měly zde být zhodnoceny výsledky získané simulacemi obou procesorů vzhledem k teoreticky očekávaným výsledkům. Tato kapitola ale v této práci chybí, protože se mi nepovedlo k jejímu zpracování vytvořit odpovídající praktické podklady. V závěrečné kapitole se objeví zhodnocení výsledků práce a budou zde nastíněny možnosti dalšího vývoje projektu.

Tato diplomová práce navazuje na projekty, které jsem vypracovala v předmětech Ročníkový projekt a Semestrální projekt, a tak je potřeba zde nastínit, jaký objem prací již byl vyřešen v rámci těchto projektů. V rámci Ročníkového projektu jsem se zabývala návrhem a implementací jednoduchého procesoru. Výsledky tohoto projektu jsem převzala do mé diplomové práce a zužitkovala jsem je zejména v kapitole čtvrté, která se přímo zabývá návrhem a implementací jednoduchého procesoru. V Semestrálním projektu bylo mým úkolem seznámit se s vybranými moderními prvky a principy, které se uplatňují v architekturách počítačů a procesorů, konkrétně se zřetězeným zpracováním instrukcí, predikcí skoků a pamětí cache. Těchto poznatků jsem využila při sestavování kapitoly třetí, kde jsou předloženy základní charakteristiky těchto prvků.

2 Základní principy počítačů a procesorů

V této kapitole budou vysvětleny některé základní pojmy, které se týkají architektury počítače a procesoru. První část bude věnována architektuře počítače. Bude zde objasněno podle [1], co chápeme pod pojmem architektura počítače, jaké základní typy architektur rozlišujeme a jaké mají jednotlivé architektury výhody a nevýhody. Vzhledem k tomu, že se v této práci budu zabývat návrhem procesoru, tedy relativně komplexního hardwarového zařízení, budu se v druhé části této kapitoly věnovat popisu základních vlastností a principů činnosti tohoto zařízení podle [2].

2.1 Architektura počítače

Pojem architektura neurčuje jednoznačné definice, schémata či principy, ale hovoří o tom, že objekt se skládá z menších částí a teprve jejich vzájemné propojení a soulad vytváří funkční celek.

Architekturu počítače chápeme jako pohled na jeho podstatné vlastnosti, které můžeme rozdělit do čtyř základních kategorií:

- struktura, uspořádání: popis jednotlivých funkčních částí a jejich propojení,
- součinnost, interakce: popis řízení dynamické komunikace mezi funkčními bloky,
- realizace, provedení: popisuje vnitřní strukturu jednotlivých funkčních bloků,
- funkcionalita, činnost: výsledné chování počítače jako celku.

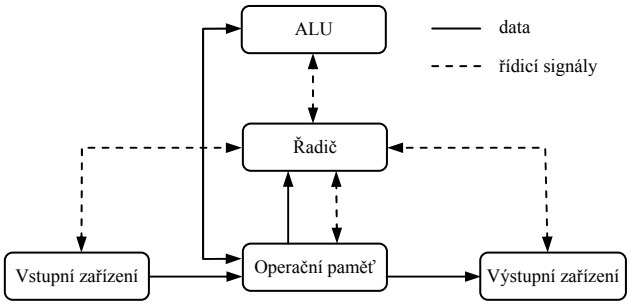
Pokud chceme porozumět základním principům činnosti počítače, musíme se zabývat všemi uvedenými hledisky.

2.1.1 Základní koncepce architektury počítače

Americký matematik maďarského původu John von Neumann a jeho kolegové z Princeton Institute for Advanced Studies na Univerzitě státu Pensylvánie ve Filadelfii definovali v roce 1945 základní koncepci počítače EDVAC (Electronic Discrete Variable Automatic Computer) a položili tak základ koncepce počítače řízeného obsahem paměti. Od té doby se sice objevilo několik různých modifikací i odlišných modelů, přesto si model von Neumannova počítače zachovává svůj význam a až na malé výjimky je jeho schéma platné dodnes.

Struktura von Neumannova počítače, jeho jednotlivé části a jejich vzájemné propojení, jsou na obr. 2.1. Činnost celého počítače řídí řídící jednotka (řadič). Řadič předává povely operační paměti, aritmeticko-logické jednotce (ALU) a vstupním a výstupním zařízením a zpět od nich dostává stavová hlášení. Řadič čte z operační paměti instrukce, dekóduje je a převádí na posloupnost signálů. Přenos

dat probíhá ze vstupních zařízení do paměti a odsud do výstupních zařízení a mezi pamětí a aritmeticko-logickou jednotkou. Aritmeticko-logická jednotka společně s řadičem tvoří procesor, jinými slovy centrální výkonnou jednotku počítače (CPU).



obr. 2.1 Základní schéma počítače podle von Neumanna

V projektu von Neumannova počítače byla stanovena určitá kritéria a principy, které musí počítač splňovat, aby byl použitelný univerzálně. Ve stručnosti je lze tyto pravidla shrnout do následujících několika bodů:

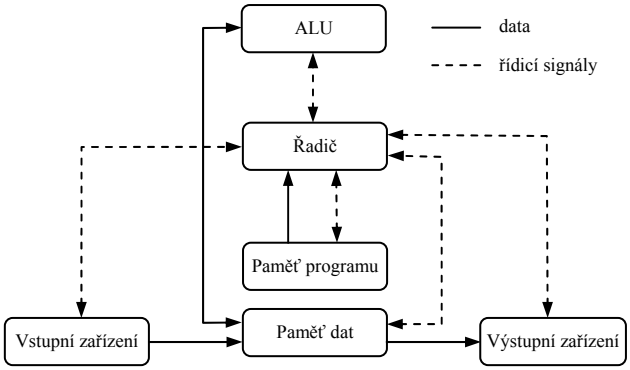
1. Počítač se skládá z paměti, řídící jednotky, aritmetické jednotky, vstupní a výstupní jednotky.
2. Struktura počítače je nezávislá na typu řešené úlohy, počítač se programuje obsahem paměti.
3. Následující krok počítače je závislý na kroku předchozím.
4. Instrukce a operandy (data) jsou v téže paměti.
5. Paměť je rozdělena do buněk stejné velikosti, jejich pořadová čísla se využívají jako adresy.
6. Program je tvořen posloupností instrukcí, ty se vykonávají jednotlivě v pořadí, v jakém jsou zapsány do paměti.
7. Změna pořadí provádění instrukcí se provede instrukcí podmíněného či nepodmíněného skoku.
8. Pro reprezentaci instrukcí, čísel, adres a znaků, se používá dvojková číselná soustava.

2.1.2 Harvardská koncepce

Několik let po vytvoření koncepce von Neumannova počítače, přišel vývojový tým odborníků z Harvardské univerzity s vlastní koncepcí počítače, která se sice od von Neumannovy příliš neliší, ale odstraňuje některé její nedostatky.

U počítačů s von Neumannovou architekturou může procesor v jednom časovém okamžiku číst resp. zapisovat buď pouze data nebo jen instrukce, což je způsobeno tím, že u této architektury je pro data a instrukce vyhrazena společná paměť a propojovací obvody. Tento nedostatek byl vyřešen

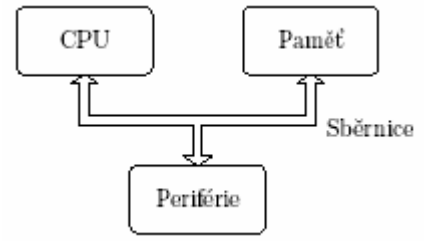
v harvardské architektuře tak, že došlo k fyzickému oddělení paměti pro data a program. Takto může procesor, díky odděleným propojovacím obvodům a separátní paměti programu a dat, zároveň přistupovat do paměti dat i paměti programu. Schéma propojení jednotlivých částí počítače v této architektuře je patrné z obr 2.2. Funkce jednotlivých funkčních bloků jsou totožné jako u počítače von Neumannova, s tím rozdílem, že řadič čte instrukce výhradně z paměti programu a výměna dat probíhá pouze s pamětí dat.



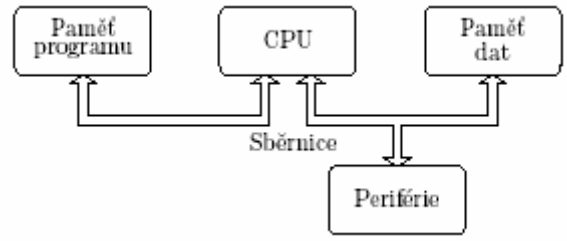
obr. 2.2 Základní schéma počítače podle harvardské architektury

2.1.3 Porovnání základních přístupů

Abychom si mohli obě koncepce porovnat, budeme vycházet ze zjednodušených schémat obou architektur. Na obr. 2.3 je schéma počítače von Neumannova typu a na obr 2.4 je schéma počítače podle harvardské koncepce.



obr. 2.3 Počítač von Neumannova typu



obr. 2.4 Harvardská architektura počítače

Základním nedostatkem obou koncepcí je sekvenční vykonávání instrukcí, které sice umožňuje snadnou implementaci systému, ale nepovoluje dnes tolik potřebné paralelní zpracování. Paralelizmy se musí simulovat až na úrovni operačního systému. Úzké místo systému je také ve sběrnicích, které nedovolují přistupovat současně do více míst paměti současně a navíc dovolují v daném okamžiku přenos dat jen jedním směrem.

Porovnání vlastností obou koncepcí:

- von Neumannova koncepce

  Výhody:

  - rozdělení paměti pro kód a data určuje programátor,
  - řídící jednotka procesoru přistupuje do paměti pro data i pro instrukce jednotným způsobem,
  - jedna sběrnice - jednodušší výroba.

  Nevýhody:

  - společné uložení dat a kódu může mít při chybě za následek přepsání vlastního programu,
  - jediná sběrnice tvoří úzké místo.

- harvardská koncepce

  Výhody:

  - oddělení paměti dat a programu přináší výhody:
    - program nemůže přepsat sám sebe,
    - paměti mohou být vyrobeny odlišnými technologiemi,
    - každá paměť může mít jinou velikost nejmenší adresovací jednotky,
    - dvě sběrnice umožňují jednoduchý paralelizmus, kdy lze přistupovat pro instrukce i data současně.

  Nevýhody:

  - dvě sběrnice kladou vyšší nároky na vývoj řídící jednotky procesoru a zvyšují i náklady na výrobu výsledného počítače,
  - nevyužitou část paměti dat nelze použít pro program a obráceně.

2.2 Architektura procesoru

Procesor je zařízení, které transformuje zadaná vstupní data podle definovaného předpisu na data výstupní. Z matematického hlediska vlastně realizuje funkci o mnoha vstupních proměnných, jejímž výsledkem je mnoho dalších proměnných výstupních. Tato transformace je definována programem, což je posloupnost řídících pokynů pro vykonání konkrétních činností, které je procesor schopen realizovat. Tyto pokyny se nazývají instrukce a činnosti (operace), které jsou jim přidělené,

jsou přesně definovány. Vstupem do procesoru budeme chápat data uložená v pamětech, kterými mohou být paměti externí, interní nebo paměti vyrovnávací (cache). Dále jím mohou být přímé, relativně izolované vstupy, jako jsou například přerušení, vstupní datové nebo komunikační porty apod. Tyto prostředky mohou být také jistým způsobem mapovány do paměti. Výstup transformační funkce bývá s množinou vstupů často totožný, popřípadě mají velkou část společnou. Výsledky se totiž také zapisují do paměti, přenášejí přes výstupní porty apod.[3]

2.2.1 Vývoj procesorů

Vývoj základní funkční jednotky počítače, procesoru, z pohledu výroby představuje přechod od diskrétních elektronických součástek, kdy počítače s takovými procesory zabíraly i celou místnost, přes integrované obvody, kdy začalo docházet k miniaturizaci takto vyráběných procesorů, až po integraci procesoru na jediný čip, a tedy vzniku mikroprocesoru. Zároveň docházelo také ke snižování výrobních nákladů při výrobě procesorů, nároků na spotřebu elektřiny, a tak se s klesajícími cenami staly počítače široce dostupné.

Vývoj procesorů v oblasti architektury se ubírá směrem k rostoucímu paralelismu vnitřních operací a provádění i vydávání instrukcí. Tento funkční vývoj lze rozdělit do třech fází podle stupně paralelismu vydávání a provádění instrukcí. První fáze je reprezentována procesory se sekvenčním vydáváním a sekvenčním prováděním instrukcí (obr 2.5), kdy doba provedení programu je dána součtem časů trvání jednotlivých instrukcí.a pohybuje se od jednotek do desítek taktů. Takové procesory nazýváme subskalární a jsou založeny na von Neumannově koncepci.

čas ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	S1	S2	S3	S4										
I2					S1	S2	S3	S4						
I3									S1	S2	S3	S4		
I4													S1	S2

obr 2.5 Sekvenční vydávání a sekvenční nezřetězené zpracování instrukcí ve čtyřech stupních S1 až S4

V dalším vývoji bylo sekvenční vykonávání instrukcí nahrazeno paralelním pomocí zřetězeného zpracování instrukcí nebo zavedením několika funkčních jednotek. U procesorů tohoto typu, nazýváme je skalární, je nadále zachováno sekvenční vydávání instrukcí a provedení programu jim v ideálním případě trvá tolik taktů, kolik je počet instrukcí, ve skutečnosti trvá déle kvůli konfliktům, o kterých se zmíním později. Příklad zpracování programu je znázorněn na obr. 2.6.

čas ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	S1	S2	S3	S4										
I2		S1	S2	S3	S4									
I3			S1	S2	S3	S4								
I4				S1	S2	S3	S4							

obr 2.6 Sekvenční vydávání a zřetězené zpracování instrukcí
ve čtyřech stupních S1 až S4

Vzhledem k tomu, že sekvenční vydávání instrukcí nestačilo zásobovat větší počet paralelně pracujících zřetězených funkčních jednotek, dospěl vývoj do třetí fáze, kterou charakterizují tzv. superskalární procesory s paralelním vydáváním a paralelním vykonáváním instrukcí. Na obr 2.7 je ukázka, jak by mohlo vypadat zpracování části programu tímto procesorem.

Další vývoj architektury se ubírá cestou soustavného zvyšování paralelismu vydávání a provádění instrukcí. Podrobněji jsou jednotlivé architektury popsány v [2].

čas ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	S1	S2	S3	S4										
I2	S1	S2	S3	S4										
I3		S1	S2	S3	S4									
I4		S1	S2	S3	S4									

obr 2.7 Ukázka vydávání více instrukcí v jednom taktu a
zřetězené zpracování ve čtyřech stupních

2.2.2 Základní koncepce architektury procesoru

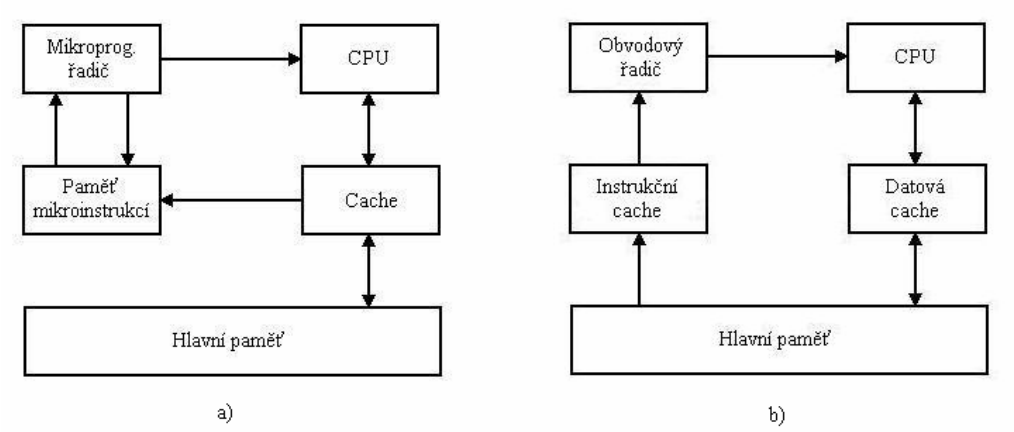
V oblasti konkrétní architektury procesoru rozlišujeme dvě základní architektonické koncepce, a to koncepci typu CISC – procesor se složitým souborem instrukcí (Complex Instruction Set Computer) a RISC – procesor s redukovaným souborem instrukcí (Reduced Instruction Set Computer). Koncepce CISC se vyvinula na základě požadavků pro podporu operací prováděných ve vyšších programovacích jazycích. Do instrukčního souboru byly přidávány stále nové operace, což vedlo k nezadržitelnému růstu složitosti těchto procesorů. Instrukce CISC používají mnoho adresovacích režimů. Pro jejich vykonávání se typicky používá mikroprogramový řadič, který je řízen mikroprogramem, kde má každá instrukce definovanou sekvenci mikrooperací. Příklad architektury CISC je na obr 2.8.

Výzkumy četnosti výskytu instrukcí ukázaly, že programátoři a kompilátory používají strojové instrukce velmi nerovnoměrně. Počet nejfrekventovanějších instrukcí je omezen na velmi úzkou část instrukčního souboru. Na základě toho se objevily pokusy o nalezení optimálního instrukčního souboru a o celkové zjednodušení struktury procesorů, čímž vznikla celá nová kategorie počítačů. Vykonávání složitých instrukcí bylo převedeno z paměti mikroprogramu do programu aplikačního a

zůstala sada jednoduchých nejčastěji používaných elementárních instrukcí. Výsledkem návrhu je počítač RISC s těmito vlastnostmi :

- v každém strojovém cyklu by měla být v ideálním případě dokončena jedna instrukce,
- mikroprogramový řadič je nahrazen rychlejším obvodovým řadičem,
- používá zřetězené zpracování instrukcí,
- počet instrukcí a způsobů adresování je malý,
- paměťové operace probíhají jen pomocí dvou instrukcí přesunů LOAD a STORE,
- instrukce mají pevnou délku a jednotný formát, který vymezuje význam jednotlivých bitů,
- používá vyšší počet registrů.

Mezi nevýhody RICS architektury patří nutný nárůst délky programů, tvořených omezeným počtem instrukcí a také díky jednotné délce všech instrukcí. Příklad architektury RISC je na obr 2.8.



obr 2.8 Příklady architektur CISC a RISC:

a) procesor CISC s mikroprogramovým řízením a společnou pamětí cache

b) architektura RISC s obvodovým řadičem a oddělenými pamětmi cache pro instrukce a data

2.2.3 Součásti procesoru

K základním prvkům procesoru patří řídící jednotka (řadič), která řídí tok programu, tedy načítání instrukcí, jejich dekódování, načítání operandů, ukládání výsledků a zpracování instrukcí. K uchování operandů a mezivýsledků slouží sada registrů. Dále procesor obsahuje jednu nebo i více aritmeticko-logických jednotek (ALU), které provádí s daty aritmetické a logické operace a některé procesory obsahují také jednu nebo několik jednotek (FPU), které provádějí operace v plovoucí řádové čárce.

Současné čipy zpravidla obsahují mnoho dalších funkčních bloků, jako paměť cache, a různých periferií, které nejsou vyloženě součástí procesoru. Vznikl proto pojem jádro procesoru, aby bylo možné odlišit vlastní procesor od integrovaných periferních obvodů. Vzhledem k tomu, že tyto obvody bývají většinou dobře sladěny s jádrem, lze je někdy chápat jako součást procesoru, a mnohdy tak dochází k rozmazání hranice mezi procesorem a počítačem.

3 Moderní prvky a principy v architekturách procesorů

V této kapitole budou rozebrány některé vybrané prvky a principy, které umožňují zlepšit výkonnost původních subskalárních procesorů. Podrobně bude popsaný princip vyrovnávací paměti (cache) s využitím [4] a [5], zřetězeného zpracování instrukcí (pipelining) a predikce skoků podle [2] a [6]. Důkladným rozborem vlastností těchto prvků se zabývám především z toho důvodu, že součástí této práce je jejich implementace do jednoduchého procesoru, jehož návrhu se budu věnovat v další kapitole.

3.1 Vyrovnávací paměť (cache)

Rychlá vyrovnávací paměť (cache) je jednou z mnoha možností, jak dosáhnout zrychlení práce výpočetního systému.

Úkolem této paměti je vytvořit mezičlánek mezi rychlým a pomalým zařízením. Jedná se o paměť typu SRAM, která je velmi rychlá, ale její cena je vyšší než u běžných RAM, proto jsou také velikosti pamětí cache mnohem nižší než u pamětí typu DRAM.

Dalším z hledisek, které nahrálo výběru paměti cache, bylo vyjádření principů časové a prostorové lokality odkazů. K formulaci principu lokality odkazů se dospělo na základě rozboru dynamické četnosti výskytu instrukcí, tedy sledováním, jak často je která instrukce programu za jeho běhu prováděna. Časová lokalita odkazů znamená, že pokud procesor použil v jistém okamžiku určitou instrukci, pak pravděpodobnost, že ji v dalším okamžiku bude provádět znovu je větší, než je pravděpodobnost použití instrukce, kterou dosud nepoužil. Prostorová lokalita odkazů se vyjadřuje v tom smyslu, že pokud bral program v poslední době instrukce z jistého bloku paměti, pak je velká pravděpodobnost, že v následující době bude brát instrukce opět z tohoto bloku. Pro data platí princip časové a prostorové lokality odkazů také, ovšem v mnohem menší míře. [Drábek]

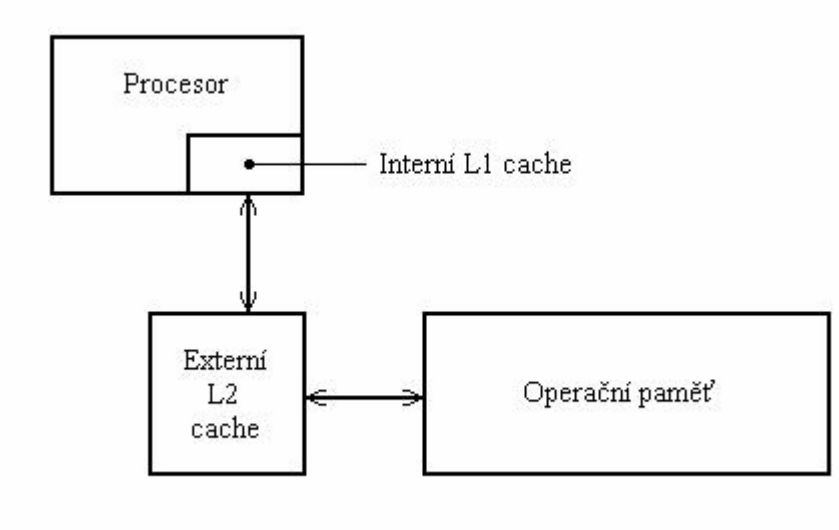
Na základě uvedených poznatků bylo schéma von Neumannovy koncepce modifikováno doplněním rychlé vyrovnávací paměti mezi rychlý procesor a pomalou operační paměť typu DRAM. V cache paměti se ukládají data pro následné využití procesorem, čímž se minimalizuje jejich přímé čtení z relativně pomalé hlavní paměti počítače a procesor má tak ve většině případů požadovaná data ihned k dispozici.

3.1.1 Hierarchie pamětí

Přidání paměti cache do architektury počítače má za následek změny v hierarchii pamětí. Nejblíže procesoru zůstávají jeho vnitřní registry, které můžeme považovat za paměť na nejnižší

úrovni. Další úroveň je nově tvořena blokem rychlých vyrovnávacích pamětí, přes které je pomocí paměťové sběrnice umožněn přístup do třetí paměťové úrovně, tedy do operační paměti. Na nejvzdálenější čtvrté úrovni je pak vnější paměť, která je k operační paměti připojena vstupněvýstupní sběrnicí.

Na úrovni pamětí cache dále můžeme rozlišit dva druhy těchto pamětí. Interní cache, také označována jako primární cache (L1 cache), je vždy integrována přímo na procesoru, má malou velikost (8 – 64 kB), slouží pro ukládání právě využívaných či potřebných instrukcí a dat a pracuje stejnou rychlostí jako procesor. Externí nebo také sekundární cache (L2 cache) dosahuje obecně vyšší kapacity než L1 cache, zpravidla 64 – 512 kB, a může být také přímo součástí procesoru nebo je umístěna na základní desce počítače. Je mezistupněm mezi primární cache a operační pamětí a obsahuje data, která procesor přímo nepoužívá, ale pravděpodobně je bude potřebovat. Pracuje v závislosti na umístění buď stejnou rychlostí jako procesor, nebo jako základní deska, přece však je čtení z této paměti stále rychlejší než čtení z hlavní paměti. Příklad zapojení interní a externí paměti cache, s externí pamětí cache umístěnou mimo procesor, je na obr 3.1.



obr 3.1 Schéma zapojení interní a externí cache paměti

3.1.2 Organizace cache paměti

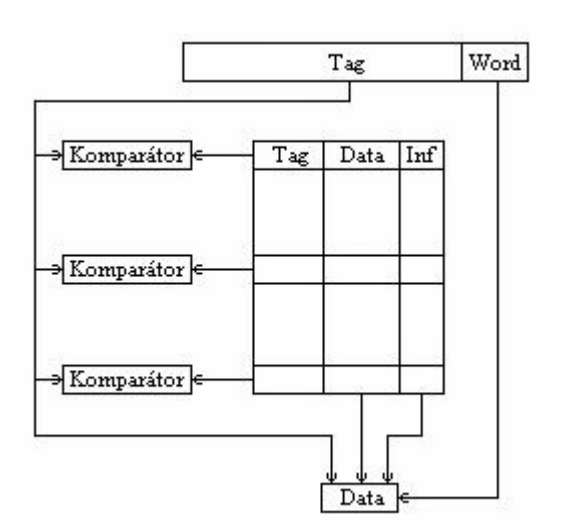
Cache paměti bývají organizovány jako paměti asociativní. Tento typ paměti je tvořen tabulkou, která vždy obsahuje sloupec, v němž jsou umístěny klíče (tagy), podle kterých se v asociativní paměti vyhledává. Dále jsou v tabulce umístěna data, která paměť uchovává, a případně další informace nutné k zajištění správné funkce paměti. Pro přístup do paměti je nutné zadat adresu, z níž data požadujeme. Tato adresa je vždy rozdělena na dvě části, nejnižší bity adresy identifikují patřičné slovo v bloku a zbývající vyšší bity specifikují samotný blok. Celá adresa bloku nebo jen její část je

považovaná za tag a porovnává se s tagy uloženými v paměti. V případě, že porovnání dopadne úspěšně, vyberou se příslušná data. Podle způsobu organizace rozlišujeme tři druhy pamětí cache:

- plně asociativní,
- n-cestně asociativní,
- přímo mapovanou.

Jednotlivé způsoby organizace paměti se tedy odlišují právě různým nahlížením na adresu bloku.

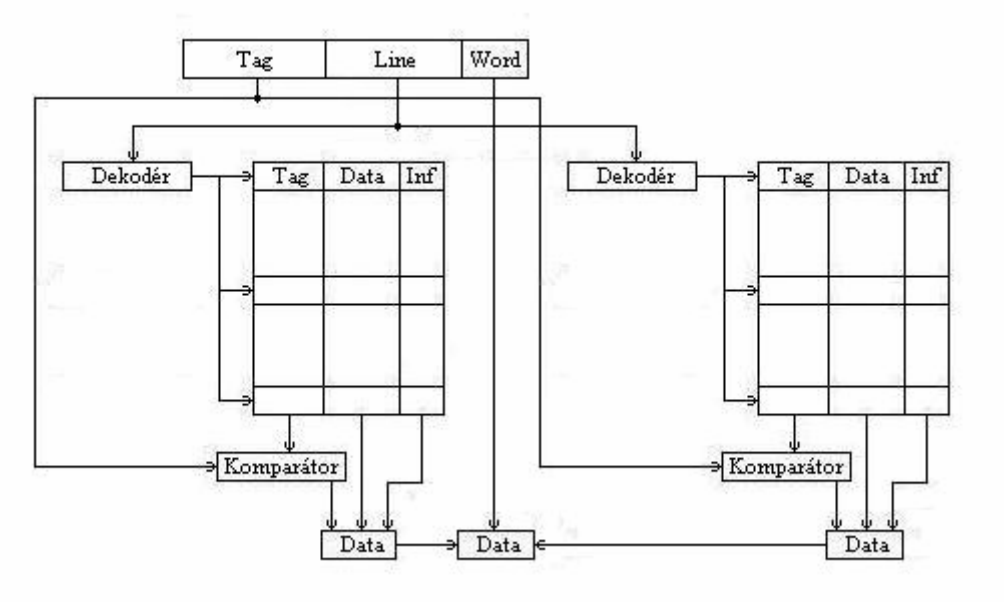
U plně asociativní cache paměti může být blok dat přečtený z operační paměti uložen do libovolné řádky cache tabulky. Princip hledání dat v tomto typu paměti je naznačen na obr 3.2. Jako tag bude sloužit celá adresa bloku. Tento tag je přiveden na vstup komparátorů, které provádí v jednotlivých řádcích porovnání tohoto tagu s tagem v daném řádku tabulky. Pokud některý z tagů v tabulce je shodný se zadaným tagem na vstupu, ohlásí odpovídající komparátor shodu a znamená to, že požadovaná informace je v cache paměti přítomna a je možné ji použít (cache hit). Pokud všechny komparátory signalizují neshodu, je to známka toho, že požadovaná informace v cache paměti není (cache miss) a je nutné ji zavést z externí paměti cache, nebo z operační paměti. Nevýhodami této organizace pamětí cache jsou nutnost použití velkého množství komparátorů a potřeba v každém řádku tabulky uchovávat celý tag, což vede k tomu, že musí mít cache paměť velkou kapacitu, do které se tyto tagy ukládají a kterou není možné využít k uchovávání dat. Z těchto důvodů se plně asociativní paměti prakticky nepoužívají.



obr 3.2 Schéma funkce plně asociativní cache

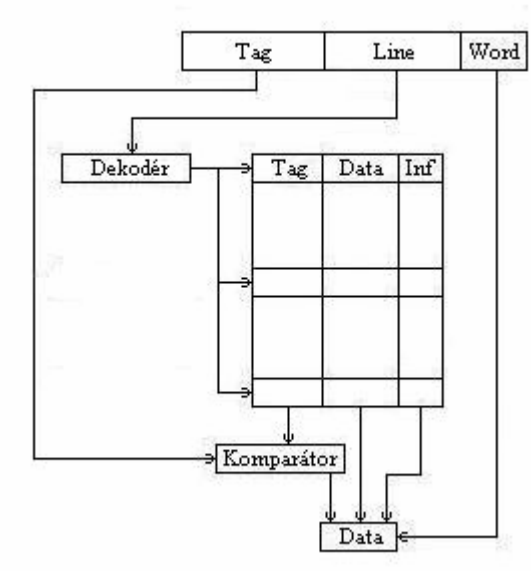
Pokud je paměť n-cestně asociativní, znamená to, že původní plně asociativní cache byla rozdělena na n tabulek se stejným počtem řádků, kde součet řádků všech n tabulek dává počet řádků původní cache, a jejichž sloupce jsou totožné s původní cache. N-cestně asociativní paměti pracují tak, že zadaná adresa bloku se rozdělí na dvě části, a to na tag a adresu třídy. Adresa třídy je přivedena na n dekodérů, které v každé tabulce vyberou jeden řádek. Z těchto řádků se potom vezmou

příslušné tagy a komparátorem se porovnají se zadaným tagem. Podobně jako u plně asociativních cache pamětí pokud jeden z komparátorů signalizuje shodu, je informace v cache paměti přítomna. V opačném případě je nezbytné informaci hledat jinde. N-cestně asociativní paměti částečně eliminují nevýhody plně asociativních cache pamětí a v současnosti jsou nejpoužívanějším typem pamětí cache. Na obr 3.3 je znázorněna n-cestně asociativní cache se stupněm asociativity n = 2.



obr 3.3 Schéma 2-cestně asociativní cache

Přímo mapovaná cache paměť je speciální případ n-cestně asociativní cache paměti pro n rovno jedné, jak je patrné z obr 3.4.



obr 3.4 Schéma funkce přímo mapované cache

Při této organizaci paměti má každý blok své pevné místo, kam může být uložen. Zadaná adresa bloku je opět rozdělena na tag a adresu třídy. Adresa třídy je přivedena na vstup dekodéru, který

podle ní vybere jeden řádek v tabulce. Tag na tomto řádku je následně porovnán se zadaným tagem, čímž se rozhodne o přítomnosti či nepřítomnosti informace v cache paměti. Výhodami tohoto přístupu jsou jednoduchý princip a nízká cena vzhledem k použití menšího počtu přídavné logiky. Nevýhodu představuje mapování bloků na pevná místa v paměti, protože pokud program potřebuje opakovaně používat bloky, které jsou mapovány do stejného místa v cache, pak bude docházet k častým výpadkům (cache miss). Přímo mapovaná cache tedy ve srovnání s n-cestně asociativní cache pamětí vykazuje nižší výkon, a proto její použití dnes není příliš časté.

3.1.3 Výměna dat

V případě, že procesor chce přečíst data z paměti, vloží na adresovou sběrnici adresu. Předáním této adresy do paměti cache se zjistí, jestli jsou zde požadovaná data umístěna. Pokud zde požadovaná data umístěna jsou, data jsou přečtena a předána procesoru. V opačném případě je nutné přečíst z operační paměti celý blok dat, jehož jsou tyto data součástí, a tento blok zapsat do paměti cache. Předtím je ale nutné v paměti cache uvolnit dostatečný prostor, aby sem mohl být tento blok zapsán. Výběr řádku, kam bude nový blok zapsán, je předem daný u přímo mapovaných cache, ale u plně asociativních a n-cestně asociativních cache se řídí jednou z následujících strategií:

- Výběr nejméně používaného řádku (LFU – Least Frequently Used Strategy),
- Výběr nejdéle nepoužívaného řádku (LRU – Least Recently Used Strategy),
- Výběr řádku, který je obsazen nejdéle (FIFO – First In First Out Strategy),
- Náhodný výběr řádku (Random Strategy).

První tři strategie vyžadují, aby každý řádek obsahoval další bity, do kterých jsou ukládány informace o využití tohoto řádku. Náhodná strategie toto nevyžaduje, jednodušeji se proto implementuje a experimentálně bylo dokázáno, že je při výběru řádku jen o něco málo méně efektivní než ostatní strategie.

V případě zápisu dat procesorem do paměti cache, se musí změny dat ve vyrovnávací paměti automaticky promítnout také v hlavní paměti, aby byla zaručena konzistence dat, což zajišťuje řadič cache. Rozlišujeme zde dva přístupy, a to přímý zápis (write-through) a opožděný zápis (write-back).

Princip přímého zápisu je jednodušší a spočívá v tom, že všechny operace zápisu do paměti cache vedou vždy také k zápisu do operační paměti. Aby nebyla zdržována komunikace mezi procesorem pamětí cache, využívá paměť cache s tímto přístupem při operaci zápisu do operační paměti pomocnou vyrovnávací paměť (buffer).

U opožděného zápisu procesor zapíše pouze do paměti cache, ale obsah operační paměti se nemění. Každý řádek cache tabulky musí tedy obsahovat bit, do kterého se zaznamená, že došlo ke změně obsahu bloku. Nastavení tohoto bitu znamená, že data uložená v tomto řádku byla změněna oproti původním datům načteným z operační paměti. V případě, že je nutné takto modifikovaný řádek odstranit z paměti cache, musí se zajistit přenesení a aktualizace tohoto bloku v operační paměti.

3.2 Zřetězené zpracování instrukcí (pipelining)

U subskalárních procesorů, jak jsem se již zmínila v kapitole 2.2.1, probíhá sekvenční vydávání a sekvenční zpracování instrukcí, což je velice časově náročné. Dále zde bylo zmíněno, že zvýšení výkonnosti je možné dosáhnout tím, že sekvenční vykonávání instrukcí nahradíme paralelním, a to pomocí zřetězeného zpracování nebo zavedením několika funkčních jednotek. V tomto textu se omezím jen na výklad o zřetězené zpracování instrukcí u skalárních procesorů, vysvětlím, jaký je jeho princip i jaké přináší výhody a nevýhody.

Zřetězené zpracování je založeno na tom, že vykonání nějaké operace se rozdělí na sekvenci nezávislých kroků, nejlépe stejného trvání, přičemž v každém kroku se používají samostatné technické prostředky. Tak je možné, že v jednom okamžiku je rozpracováno několik operací, každá je ovšem v jiném stavu rozpracování a využívá jiný stupeň řetězu technických prostředků. Zřetězená implementace tedy předpokládá neustálý přísun údajů, nad nimiž se provádí stejná základní operace. Tuto operaci musí být dále možné rozdělit na sekvenci nezávislých kroků realizovaných jednotlivými stupni řetězu s tím, že trvání jednotlivých kroků by mělo být přibližně stejné.

3.2.1 Linky pro zřetězené zpracování instrukcí

Nejstarší typ linky pro zřetězené zpracování instrukcí měl jen dva stupně, instrukční jednotku (F – Fetch), která načítá instrukce z paměti, provádí určení typu instrukce a výpočet adresy, a prováděcí jednotku (E – Execute), která načítá operandy, vykonává operaci a ukládá výsledek. Rozdílné doby zpracování v obou stupních, kdy zpracování v prováděcí jednotce trvalo mnohem delší dobu než v jednotce instrukční, vedly k tomu, že provádění se rozdělilo na dva stupně, dekódování (D – Decode), v němž se provádí i načtení registrových operandů, a vlastní provádění (E). V dnešní době obsahuje většina instrukčních linek 5 – 12 stupňů.

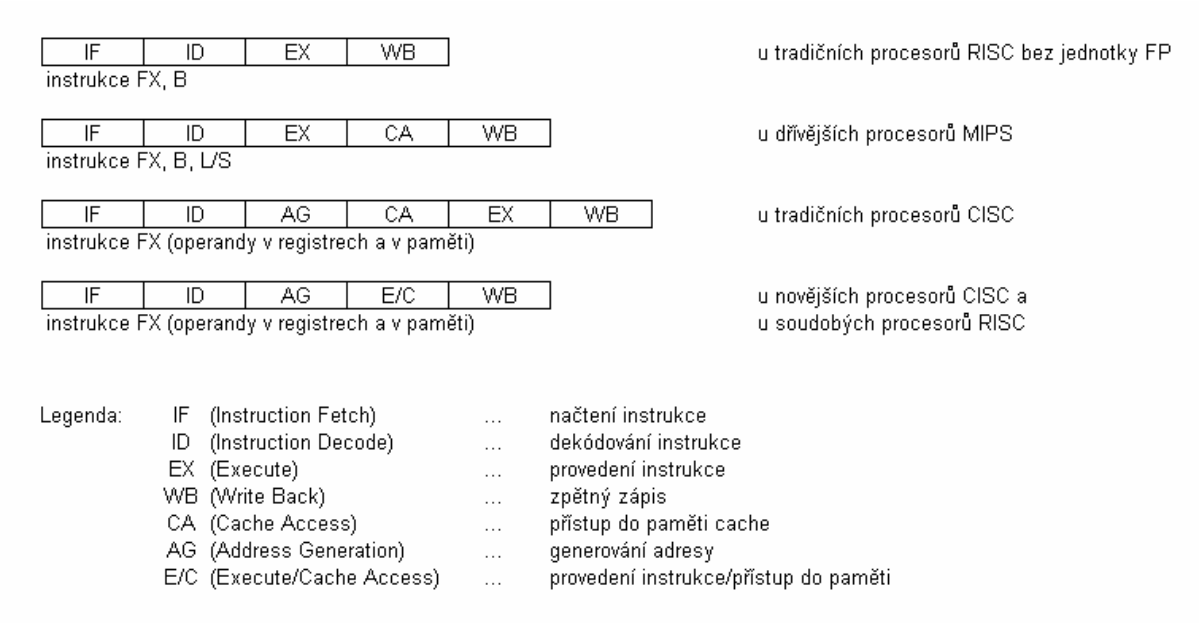
Řešení instrukční linky záleží na tom, jak je implementováno zřetězené zpracování jednotlivých skupin instrukcí. V instrukčních souborech lze rozpoznat zhruba čtyři hlavní podmnožiny instrukcí:

- instrukce pro aritmetické a logické operace s operandy s pevnou řádovou čárkou (FX – Fixed Point),
- instrukce pro operace v plovoucí řádové čárce (FP – Floating Point),
- instrukce pro operace skoků (B - Branching),
- instrukce pro operace s daty prováděné nad pamětí (L/S – Load/Store).

Některé způsoby řešení instrukční linky, které se používají nebo se používaly dříve, jsou uvedeny na obr 3.5.

Linka pro zřetězené zpracování instrukcí jsou u procesorů většinou synchronní, řízená hodinovým signálem. Celá linka je rozdělena na stupně, pokud možno s přibližně stejným zpožděním,

mezi které jsou vloženy oddělovací registry pro zachycení mezivýsledků. Zápis výsledků jednotlivých stupňů do příslušných záchytných registrů je pak synchronizován hodinovým signálem, jehož perioda je určena nejpomalejším stupněm. Linka je tedy sestavena tak, že v každém taktu je celý stav zpracování instrukce obsažen úplně a výhradně v obvodech jen jednoho stupně, což znamená, že ostatní stupně jsou volné a dají se použít pro zpracování dalších nezávislých instrukcí. Pokud nastane situace, kdy zpracování instrukce nemůže pokračovat kvůli závislosti na některé předchozí instrukci, řeší se tato situace technicky vložením prázdného taktu a nebo vložením instrukce NOP (No OPeration) kompilátorem do kódu programu.



obr 3.5 Základní přístupy k řešení zřetězeného zpracování instrukcí

3.2.2 Konflikty při zřetězeném zpracování instrukcí

U zřetězeného zpracování je cílem, aby v každém taktu byla dokončena jedna instrukce, což není v praxi dosažitelné. Důvodem je to, že instrukce na sobě často závisí takovým způsobem, že určitá instrukce nemůže být provedena, dokud není vykonána jedna nebo více předchozích instrukcí. Existují celkem tyto tři typy závislostí mezi instrukcemi:

- datové (údajové) závislosti lze rozdělit do třech typů:
  RAW (Read After Write, čtení po zápisu),
  WAR (Write After Read, zápis po čtení) a
  WAW (Write After Write, zápis po zápisu)
- řídící závislosti – se týkají instrukcí prováděných po podmíněném skoku
- strukturní závislosti (prostředků) – vznikají, když instrukce vyžaduje prostředek, který je ještě používán předchozí instrukcí

Ne všechny závislosti se při zřetězeném zpracování musejí projevit jako konflikty. Některé závislosti může odstranit již kompilátor při překladu programu třeba přeskládáním instrukcí, které nebude mít vliv na jejich výsledky. Řadu konfliktů lze také potlačit až za běhu programu technickými prostředky.

Datové závislosti typu RAW vznikají v důsledku postupu zpracování dat v registrech a v paměti a označují se jako postupové závislosti (flow dependencies). U těchto typů závislosti se zpracování jedné instrukce zastaví ve fázi dekódování kvůli tomu, že tato instrukce ke svému zpracování potřebuje data, která jsou modifikována některou z předchozích instrukcí. K dalšímu zpracování této instrukce může dojít až v době, kdy bude mít k dispozici platná data. Zkrácení doby čekání umožňuje technika předávání dat (forwarding, bypassing), díky které lze předat výsledek čekající instrukci již z výstupu aritmeticko-logické jednotky nebo výstupu datové paměti. Tato technika umožňuje sice zkrátit dobu čekání, ale nedokáže vždy úplně odstranit tyto závislosti, takže můžeme tyto závislosti označit za pravé. Snížení počtu konfliktů lze dosáhnout plánováním posloupnosti instrukcí kompilátorem.

Závislosti typu WAR, nazývané také protiproudé závislosti (anti-dependencies), mohou vzniknout při dokončování instrukcí v jiném pořadí, než jsou uvedeny v programu. Toto může nastat u různě dlouhých operací s pohyblivou řádovou čárkou nebo u superskalárních procesorů prováděním instrukcí mimo pořadí. Tyto závislosti mohou být vždy odstraněny, a proto se označují jako nepravé závislosti.

Poslední skupinou datových závislostí jsou závislosti WAW, jinak nazývané výstupní závislosti (output dependencies), které se u operací s pevnou řádovou čárkou nevyskytují, ale často se s nimi setkáme u operací s pohyblivou řádovou čárkou. Příkladem může být provádění dvou instrukcí, z nichž první provádí násobení a druhá sčítání a obě zapisují výsledek do stejného registru. Pokud bude násobička i sčítačka pracovat nezávisle, pravděpodobně se stane, že výsledek sčítání bude zapsán dříve než výsledek násobení a obsah registru tak bude neplatný. Je proto nutné pozdržet zápis výsledku druhé instrukce až po zapsání výsledku instrukce první. I tento typ závislostí lze vždy odstranit a jde tedy o nepravé závislosti.

Nepravé závislosti typu WAR a WAW lze vždy odstranit a metoda, která se k tomu používá, se nazývá přejmenování registrů. To může být realizováno buď staticky kompilátorem nebo dynamicky přímo v procesoru.

Druhou kategorií závislostí mezi instrukcemi jsou řídící závislosti (control dependencies). Tyto závislosti se projevují při provádění instrukcí skoků. Jde o to, že instrukce prováděné po podmíněném skoku, závisejí na výsledku testu podmínky, který je součástí skokové instrukce. Pro správné pokračování podmíněného skoku je třeba znát výsledek testu podmínky a cílovou adresu, kam se má skočit. Abychom tyto údaje znali co nejdříve a nedocházelo tak v případě provedení skoku chybně k vykonáváním instrukce, která po skoku následuje, je potřeba vypočítat adresu skoku pomocí samostatné sčítačky již ve fázi dekódování a v této fázi také vyhodnotit podmínku skoku. Pokud se zjistilo, že se skok má provést, bude již v dalším taktu správně načtena instrukce z cílové adresy

skoku a to s pokutou 1 takt. V případě, že test podmínky byl negativní, doběhne instrukce, která následovala po instrukci skoku a která byla načtena v době dekódování instrukce skoku, do konce bez pokuty a pokračuje se zpracováním instrukcí na následujících adresách. Aby se na správné adrese pokračovalo s minimálním zdržením používají se různé metody predikce, pomocí kterých se předpovídá, jestli se skok provede či nikoliv.

Poslední kategorií závislostí mezi instrukcemi jsou závislosti prostředků (resource dependencies), nebo také strukturní závislosti. Tím rozumíme situace, kdy instrukce vyžaduje prostředek, který je ještě používán předchozí instrukcí. Typickými příklady jsou použití nezřetězené jednotky pro dělení v procesoru, kdy ji chtějí využívat po sobě následující instrukce dělení, nebo požadavek zápisu do registrů dvěma instrukcemi ve stejném taktu, který lze ale řešit použitím dvou nebo více bran pro zápis. Další konflikt, který je ale řešitelný, je souběžný přístup do společné paměti pro instrukce a data, když se má zároveň číst instrukce a číst nebo zapisovat data. Tento konflikt je možné vyřešit oddělením paměti pro data a instrukce (harvardská koncepce 2.1.2). Celkově jsou závislosti prostředků řešitelné replikací jednotek (sběrnic, bran, funkčních jednotek).

3.3 Predikce skoků

Moderní procesory mají jednotku pro predikci skoků. Tato jednotka si uchovává informaci o jednou použitých skokových instrukcích a na základě těchto informací je schopna s předstihem, již ve fázi dekódování, predikovat, zda bude skok proveden a pokud bude, tak na jakou adresu. Na základě této predikce procesor načte a začne zpracovávat tu instrukci, která bude pravděpodobně následovat.

3.3.1 Pevná predikce

Implementačně nejjednodušší je prediktor s tzv. pevnou negativní predikcí skoku, který předpokládá, že se skok vždy neprovede. Tento prediktor funguje následovně. Než se ve fázi dekódování zjistí, že jde o instrukci skoku, načte se již další instrukce I_1 z následující adresy. Pokud je splněna podmínka skoku, je načtena instrukce I_2 z cílové adresy skoku s pokutou 1 takt a k provedení předchozí načtené instrukce I_1 nedojde. V případě, že test podmínky je negativní, dojde k provedení načtené instrukce I_1 a pokračuje se zpracováním instrukce na následující adrese.

Implementace tohoto prediktoru je sice jednoduchá, ale často se platí pokuta za špatnou predikci, protože skoky se mnohem častěji provedou než neprovedou. Řešením se zdá být opačná predikce, ale v tom případě by musela být cílová adresa skoku známa před vyhodnocením podmínky skoku, a to co nejdříve po fázi výběru instrukce skoku, aby se dalo pokračovat na nové cestě s minimálním zdržením. Pevná pozitivní predikce se proto používá jen výjimečně.

3.3.2 Skutečná predikce

Vedle pevné predikce existuje ještě predikce skutečná se dvěma možnými výsledky. Pokud predikce závisí jen na kódu programu, potom mluvíme o statické predikci. Při tomto typu predikce jsou předem určena nějaká pravidla, která určují nastavení pozitivní resp. negativní predikce. Vztahují se většinou na některou z vlastností instrukce, jako je

- operační znak,
- směr skoku,
- predikce kompilátorem, která je uložena ve zvláštním bitu skokové instrukce při kompilaci.

Při použití predikce kompilátorem se dosahuje lepší přesnosti predikce použitím profilace, tedy doplněním bitu predikce až při rekompilaci podle statistiky chování skoku při několika předchozích běžích programu s různými vstupy. V ostatních případech nebývá tato predikce příliš přesná. Pokud však závisí predikce na dosavadní historii běhu programu, jde o dynamickou predikci, která je sice složitější, ale dosahuje vysoké přesnosti. Procesor zde bere v úvahu historii zpracování určitého počtu předchozích skoků, o které musí vést záznam. Historie dosavadního zpracování instrukcí je zde reprezentována

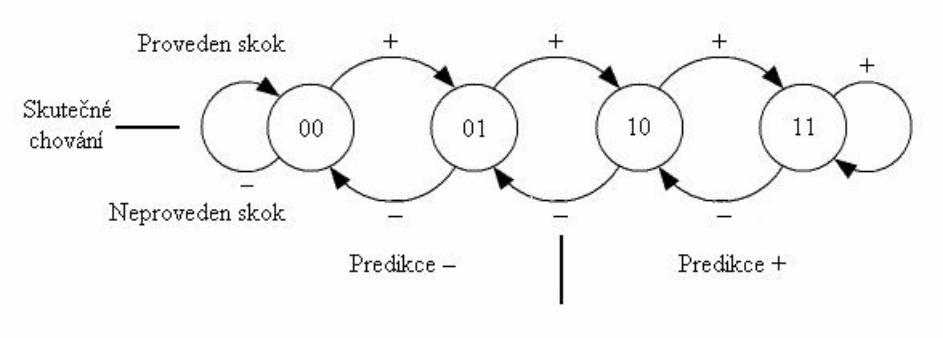
- jedním až třemi bity u každého skoku ve zvláštní tabulce historie skoků BHT (Branch History Table), případně v instrukční paměti,
- existencí záznamu ve zvláštní paměti cache predikovaných cílových adres skoků BTAC (Branch-Target Address Cache) nebo přímo cílových instrukcí skoků BTIC (Branch Target Instruction Cache).

Dynamická predikce využívá technické prostředky a na základě předchozích průchodů skoky provádí predikci jejich budoucího chování.

Nejjednodušším představitelem dynamické predikce je jednobitový prediktor skoků, který používá pro tabulku historie skoků malou paměť adresovanou několika nejnižšími bity adresy skokové instrukce (8 – 12 bitů). V paměti je na každé adrese uložen jeden bit, který udává, zda se v předchozím průchodu skok provedl nebo neprovedl. Ve fázi dekódování, když se zjistí, že se jedná o instrukci skoku, se podle tohoto bitu zvolí adresa pro načtení další instrukce. Po vypočtení podmínky skoku, v případě nesprávné predikce, se bit na odpovídající adrese změní na opačný, proto lze tento prediktor chápat jako saturovaný čítač. Jeho použití pro snížení latence skoku má smysl jen v případě, že je cílová adresa skoku známa o jeden či více taktů dříve než výsledek podmínky skoku. Nevýhodou je nesprávná predikce při první a poslední iteraci smyčky, vnořené do jiné smyčky.

Přesnější predikce dosáhneme použitím dvoubitového prediktoru, který pracuje na principu saturovaného binárního čítače. Stav čítače je inkrementován nebo dekrementován v závislosti na tom, jestli se skok ve skutečnosti provede či nikoliv, a čítač saturuje ve stavu 00 nebo 11. K uložení dvoubitové historie skoků používá malou paměť cache, která je adresovaná adresou skokové instrukce, a dovoluje tak nahlédnout do tabulky už ve fázi výběru instrukce, kdy ještě nevíme, zda jde

o instrukci skoku. Pokud je adresa v tabulce nalezena, jde o skok a adresa pro načtení další instrukce se zvolí podle predikce. Pokud instrukce byla skok a v tabulce dosud zaznamenána nebyla, vytvoří se nový záznam a počáteční stav se nastaví buď na hodnotu 11 nebo třeba podle bitu predikce nastaveného kompilátorem. Po zpracování skoku se stav v tabulce aktualizuje podle skutečného chování, jak je zobrazeno v přechodovém diagramu na obr 3.6. Mezi vícebitovými prediktory se dvoubitové prediktory uplatňují nejčastěji kvůli dost vysoké přesnosti, přiměřené složitosti a nízké latenci predikce.



obr 3.6 Diagram přechodů 2-bitového prediktoru skoků

Další typy predikce zaznamenávají do paměti cache jako klíče adresy skokových instrukcí, které se vyskytly v poslední době, a k nim odpovídající cílové adresy skoků u BTAC, nebo přímo cílové instrukce u BTIC. Jejich cílem je snížit pokutu při provedení skoku. Pomocí BTAC dokážeme redukovat tuto pokutu u nepodmíněných skoků na nulu. Pokud se do BTAC zaznamená adresa, která předchází nepodmíněnému skoku, je možné realizovat nepodmíněný skok dokonce za nulovou dobu, tím že se instrukce skoku přeskočí a načte se až instrukce z cílové adresy skoku. BTAC lze využít i pro zrychlení podmíněných skoků tak, že se sem poznamenají jen uskutečněné skoky a neprovedené skoky se vyřadí. Technika BTAC je implementovaná v řadě moderních procesorů někdy i v kombinaci s dvoubitovou predikcí, ale jejím problémem je asociativní přístup, který je nákladný a omezuje kapacitu paměti BTAC. Predikce s pamětí cache BTIC se používá v případě, že je pokuta při provedení skoku vysoká vlivem hodně dlouhého přístupu do instrukční paměti cache. Řešení spočívá v tom, že se do paměti cache BTIC místo cílových adres skoků ukládají přímo cílové instrukce. Predikce funguje stejně jako u BTAC. Význam BTIC bude asi upadat se zkracováním přístupu do instrukční paměti cache.

4 Návrh jednoduchého procesoru

Obsah této kapitoly byl vytvořen v rámci Ročníkového projektu a s drobným úpravami byl převzat do této diplomové práce, abych zde mohla nastínit jeden z možných způsobů, jak postupovat při návrhu jednoduchého procesoru. Navržený procesor bude základním kamenem této práce a až jeho výsledný návrh bude rozšiřován o další prvky, jejichž popisu se budu věnovat v dalších kapitolách.

Nejprve si obecněji popíšeme některé skutečnosti, kterými je nutné se při návrhu zabývat. Před úvahami nad dekompozicí procesoru na jednotlivé části bude vhodné se zamyslet nad uspořádáním pamětí procesoru a uspořádáním ústředního výpočetního systému. Dále se budeme zabývat návrhem jednotlivých částí procesoru, ze kterých potom sestavíme celý procesor.

4.1 Paměti procesoru

Rozdělení operační paměti procesoru lze provést dvěma možnými způsoby. Jak jsem se již zmínila v kapitole 2.1, existují dva směry, kterými se lze ubírat. V případě von Neumannovy koncepce je u procesoru k dispozici jedna hlavní paměť, která obsahuje jak program (instrukce), tak data programu. Při tomto uspořádání je třeba určit, která část paměti má být interpretována jako datová a která jako program. Je tedy třeba určit místo, kde bude program v paměti začínat. Prolínání dat a programu v jedné paměti umožňuje programátorovi měnit kód prováděného programu přímo za běhu, což rozšiřuje možnosti programování, na druhé straně je tu potenciální nebezpečí, že uděláme v kódu programu chybu, a pak k těmto změnám může dojít neúmyslně.

V případě Harvardské koncepce má procesor k dispozici dvě navzájem oddělené paměti. Datová paměť určená k ukládání dat, se kterými program manipuluje, umožňuje přístup jak pro čtení tak pro zápis. Narozdíl od toho paměť programu, ve které je uložen pouze kód programu, je dostupná pouze pro čtení. Zápis do paměti programu není za běhu programu možný, čímž odpadá možnost nedobrovolné modifikace programu v průběhu provádění. Při tomto uspořádání jsou také navzájem odděleny datové cesty, kterými jsou vedena instrukční či datová slova.

Vzhledem k možnosti určit různé velikosti pro paměť dat a paměť programu a též kvůli přehlednější manipulaci s daty v simulačním programu jsem v mém návrhu zvolila rozdělení pamětí podle Harvardské koncepce.

4.2 Organizace centrálního výpočetního systému

Toto je druhý problém, kterým se musíme zabývat před samotným návrhem. Je totiž nutné definovat zdroje dat a cíle výsledků pro výpočetní (aritmeticko – logickou) jednotku. Existují tři možná řešení,

jejichž principy si nyní stručně vysvětlíme a zvolíme si to řešení, které nám bude nejvíce vyhovovat pro náš návrh procesoru.

4.2.1 Registrová architektura

K dispozici je konečný počet registrů, ve kterých jsou uloženy zdrojové operandy operace a zároveň se do jednoho z registrů uloží také výsledek operace. Každá operace může ke své činnosti využívat libovolné registry. Registrová architektura vede k jednoduššímu programování vzhledem k dostatečnému počtu registrů, které umožňují ukládat jednotlivé proměnné výpočtu lokálně. Přístup k registrům je také časové méně náročný než přístup do paměti. Nevýhodou této architektury je nutnost explicitně uvádět identifikace registrů v instrukci, což vede k nárůstu délky instrukčního slova.

4.2.2 Akumulátorová architektura

Podobá se registrové architektuře tím, že máme k dispozici opět sadu registrů, ovšem počet registrů u akumulátorové architektury bývá nižší než u registrové. Jeden ze zdrojových operandů je vždy určen implicitně a je umístěn v tzv. akumulátoru, což je také registr, ale má speciální význam a zapojení. Do akumulátoru je uložen také výsledek operace. Význam použití tohoto registru je snaha zmenšit část instrukčního slova potřebného k adresaci operandů. Při použití akumulátoru bude tedy potřeba adresovat pouze jeden zdrojový registr. Tato architektura má nevýhodu v tom, že program je potřeba psát tak, aby byl před každou operací jeden zdrojový operand v akumulátoru a zároveň počítat s akumulátorem jako výsledkem, což může učinit program složitějším a poněkud méně přehledným.

4.2.3 Zásobníková architektura

Je nejefektivnější výpočetní architektura, pokud se jedná o využití hardwarových zdrojů. Využívá zásobník pro uložení operandů i výsledku. Postup vykonání operace je takový, že se nejprve z vrcholu zásobníku vyjmou dva operandy, poté je s nimi provedena operace a výsledek je uložen zpět do zásobníku. Tento způsob je realizací tzv. postfixové notace, kdy operační znaménko následuje až za uvedením zdrojových operandů. Přestože se zdá, že akumulátorová architektura je přijatelným kompromisem mezi šířkou potřebnou k adresaci operandů a srozumitelností či složitostí programu, programátorovi může být zátěží při psaní programu právě kvůli používání postfixové notace.

V mém projektu budu používat registrovou architekturu, kvůli pohodlnějšímu a přirozenějšímu zápisu programu.

4.3 Datová šířka

Dalším úkolem je stanovit, s jakou šířkou dat se bude operovat uvnitř procesoru. Při volbě šířky instrukčního slova existuje dvě možnosti. Buď zvolíme pevnou šířku instrukčního slova, tedy všechny instrukce budou zabírat stejný paměťový prostor, nebo bude šířka instrukce různá, určená dle potřeby. Pevná šířka instrukce znamená jednodušší načítání a dekódování instrukce a zároveň je zpracování instrukcí uniformní a zabírá stejný časový úsek. Naproti tomu proměnná šířka instrukčního slova vede k optimálnímu využití paměti, ale na druhou stranu načítání celé instrukce zabere různou dobu a délka vykonání instrukce je závislá na délce jejího operačního kódu, což komplikuje optimalizaci zpracování instrukcí, pokud chceme provést jejich rozfázování a zřetězené zpracování.

Kvůli jednodušší implementaci a výhodám při zpracovávání instrukcí, jak z hlediska sekvenčního zpracování, tak při zřetězeného zpracování, jsem zvolila při návrhu využít pevnou šířku instrukčního slova. Instrukční slovo tedy bude zahrnovat operační kód instrukce a v závislosti na instrukci identifikaci operandů a výsledku. Vzhledem k tomu, že instrukční sada bude obsahovat podmíněné a nepodmíněné skokové instrukce, je nutné u těchto instrukcí zajistit, aby v instrukčním slově byla obsažena adresa odkazující do paměti programu. Dodáme ještě, že všechny navržené instrukce budou nést veškeré potřebné informace nutné k provedení příslušné operace a tudíž se příslušná operace v procesoru vykoná atomicky, to znamená, že žádná instrukce nebude k vykonání příslušné operace potřebovat předchozí provedení nějaké jiné instrukce.

Konkrétní šířku instrukčního slova zvolíme dále při návrhu instrukčního souboru v závislosti na velikosti datové paměti, počtu registrů, velikosti konstant a typech a počtu instrukcí.

4.4 Návrh kódování instrukcí

V této části se budeme zabývat konkrétním rozvržením instrukcí, rozčleněním instrukcí na základě podobných vlastností a také, zde zvolíme konkrétní šířku instrukčního slova, kterou budeme používat.

Nejprve se zamyslíme nad otázkou, které operace by měl vlastně navrhovaný procesor umět provádět a jaké instrukce k tomu bude potřebovat. Protože se má jednat o jednoduchý procesor, postačíme se souborem základních běžně používaných instrukcí, které si nyní rozdělíme do několika skupin.

- Pro základní početní operace bychom měli mít k dispozici instrukce, které budou realizovat operace součtu a rozdílu. Pro počítání s většími čísly, než které bude možno uložit v registrech, rozšíříme tyto operace ještě o možnost brát při výpočtu v úvahu přenos z nižšího řádu.
- Další instrukce, které by neměli v instrukčním souboru chybět, jsou instrukce, které umožní provádět základní operace logické. Tedy potřebujeme instrukce pro provedení

logických operací and, or a xor na bitové úrovni. Operace not je nahrazena použitím operace xor.

- Další skupinou v oblasti logických operací mohou být instrukce, které realizují operace logických posunů a rotací, které nám umožní provádět omezené operace celočíselného násobení a dělení mocninami dvou.
- Dále potřebujeme instrukce, které nám umožní provádět přesuny dat mezi jednotlivými registry a také vkládání konstanty do registru.
- Pro operace nad datovou pamětí potřebujeme instrukce, které poskytnou možnost zapsat data do paměti a přečíst data z paměti.
- Možnost interakce procesoru s okolím zajistí instrukce pro vstupně/výstupní operace, které umožní zapsat data na výstupní sběrnici nebo přečíst data ze vstupní sběrnice.
- Aby bylo možné psát složitější programy s podmínkami, k tomu nám poslouží instrukce skoků. Budeme používat instrukci nepodmíněného skoku a instrukce skoků na základě splnění podmínky, tzv. podmíněné skoky.
- Pro podporu podprogramů v rámci programu poslouží instrukce, ukládání a načítání návratových adres.

Instrukční soubor je samozřejmě možné nadále rozšiřovat např. o instrukce pro operace násobení, dělení a mnoho dalších, ale pro tento jednoduchý procesor vystačíme zatím s výše navrženými skupinami instrukcí.

Tyto skupiny ještě rozdělíme podle typu a počtu operandů.

- Instrukce se dvěma zdrojovými operandy a jedním cílovým – všechny instrukce aritmetické a instrukce pro logické operace and, or, xor.
- Instrukce s jedním zdrojovým a jedním cílovým operandem – instrukce přesunu mezi registry, instrukce pro práci s pamětí a instrukce pro vstupně/výstupní operace.
- Instrukce s jedním zdrojovým a zároveň cílovým operandem - instrukce pro logické posuny a rotace
- Instrukce s jedním zdrojovým operandem – instrukce skoků a instrukce volání podprogramu, jejichž operandem je adresa do paměti programu.
- Instrukce bez operandů – instrukce návratu z podprogramu.

Nyní máme tedy představu o tom, jak by asi ta která instrukce měla vypadat, takže můžeme začít přemýšlet o samotném kódování instrukcí. Jak jsme se již rozhodli, každá instrukce ponese ve svém instrukčním kódu úplnou informaci potřebnou k provedení příslušné operace. Největší nároky na šířku instrukčního slova budou mít instrukce z první skupiny v rozdělení podle počtu operandů, protože potřebujeme v instrukčním slově vyhradit místo pro tři adresy jednotlivých operandů.

Protože už víme, podle jakých kriterií je vhodné se orientovat při volbě šířky a velikosti jednotlivých částí navrhovaného systému, zvolíme si nyní datovou šířku, a stanovíme ji na 8 bitů. Tedy veškeré konstanty a data uložená v registrech budou mít maximálně tuto šířku.

Nyní se postupně rozhodneme pro nejvhodnější šířku instrukčního slova. Zvolíme si v rámci možností, co nejvyšší počet registrů, což nám umožní pohodlnější programování používáním většího prostoru pro lokální proměnné. Pokud zvolíme například čtyřbitový identifikátor registru, tak budeme mít k dispozici 16 registrů. Spočítejme šířku instrukčního slova pro nejvíce náročné instrukce se dvěma zdrojovými a jedním cílovým operandem takovými, že cílový je vždy uložen do registru a zdrojovými můžou být hodnoty v registrech nebo přímý operand (konstanta). V tomto případě by byla šířka části instrukčního slova potřebná k adresaci 8+4+4 = 16 bitů. Pro ostatní instrukce by byla šířka instrukčního slova 16 bitů dostačující. Řešením se zdá být zvětšit šířku instrukčního slova, ale potom by pro většinu instrukcí zůstala velká část slova nevyužitá. Pokusíme se tedy nějakým způsobem dosáhnout šířky instrukčního slova 16 bitů. Abychom nemuseli zrušit možnost používat přímé operandy v aritmetických a logických operacích, využijeme principu akumulátorové architektury, prohlásíme první zdrojový operand instrukce zároveň operandem cílovým a tímto krokem ušetříme 4 bity, které u těchto instrukcí využijeme pro operační kód instrukce. Pokud nyní spočítáme maximální šířku potřebnou k adresaci v instrukčním slově, zjistíme, že pro všechny typy instrukcí je dostačující šířka 16 bitů. U všech instrukcí budou 4 bity použity pro operační kód instrukce a další bity budou rozděleny podle typu instrukce. Instrukce s dvěma operandy v registrech budou využívat 8 bitů pro adresaci těchto registrů s tím, že první z nich bude také registrem cílovým a zbývající 4 bity budou použity pro rozlišení operací v rámci jedné instrukce. Pokud instrukce využívá jen jeden registr, jako je tomu u instrukcí posuvů a rotací, pak potřebuje pouze 4 bity na adresaci tohoto registru a ostatní bity slouží jako pomocné. U skokových instrukcí a instrukcí přesunů používáme přímý osmibitový operand, který funguje jako adresa do programové , resp. datové paměti, a zbývající 4 bity představují u instrukcí přesunů adresu zdrojového nebo cílového registru a u skokových instrukcí slouží pro vzájemné rozlišení operací v rámci jedné instrukce. Velikost přímého operandu nám určila kapacitu datové a programové paměti na 256 záznamů. Na obr. 4.1 až obr. 4.4 si názorně ukážeme všechny možnosti adresace v instrukčním slově. Úplný instrukční soubor, se kterým umí procesor pracovat je popsán v příloze A.

operační kód instrukce	adresa registru op1/cíl	adresa registru op2	pomocné bity
O O O O	R R R R	R R R R	X X X X

obr 4.1 Adresace u instrukcí se dvěma operandy v registrech

operační kód instrukce	adresa registru op1/cíl	konstanta	
O O O O	R R R R	K K K K	K K K K

obr. 4.2 Adresace u instrukcí s jedním operandem v registru a konstantou

operační kód instrukce	adresa registru op1/cíl	pomocné bity	
O O O O	R R R R	X X X X	X X X X

obr. 4.3 Adresace u instrukcí s jedním operandem v registru

operační kód instrukce	pomocné bity	adresa	
O O O O	X X X X	A A A A	A A A A

obr. 4.4 Adresace u instrukcí s adresou

4.5 Komponenty procesoru

Protože implementuji procesor a jeho části v jazyce VHDL a převážně používáním behaviorálního popisu, struktura komponent zde nebude prezentována, naproti tomu zde bude popsána funkce každého bloku se schématem zobrazujícím jeho vstupy a výstupy.

4.5.1 Programový čítač

Programový čítač je důležitým prvkem procesoru, protože určuje, ze které adresy v paměti programu se bude načítat instrukce.

Tento mechanismus funguje tak, že vloží na adresovou sběrnici adresu do paměti programu, ze které se načte následující instrukce, a zároveň si tuto adresu uloží do pomocného registru. Po načtení instrukce provede programový čítač inkrementaci adresy uložené v pomocného registru. V našem případě procesor pracuje také s instrukcemi skoku a instrukcemi podporujícími skoky do podprogramů, takže je třeba programový čítač rozšířit o schopnost reagovat správně na tyto instrukce. Dosáhneme toho tak, že rozšíříme čítač o možnost paralelního vstupu nové adresy uložené v instrukci skoku, nebo v případě návratu z podprogramu vstupu adresy ze zásobníků návratových adres. Toto rozšíření má za následek, že musíme na vstup čítače přidat signál, kterým budeme řídit, zda se má inkrementovat původní adresa, nahrát adresa skoku nebo načíst adresa ze zásobníku návratových adres. V případě podmíněných skoků nahrajeme adresu z programu pouze v případě, že je splněna

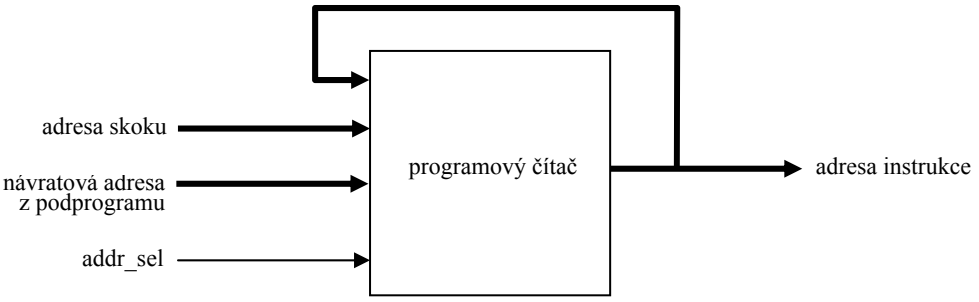
nějaká podmínka, závislá na typu skokové instrukce. Nastavení signálů v závislosti na jednotlivých instrukcích je vidět v tab 4.1.

instrukce	addr_sel	inkrementace adresy	příznakové registry	
			zero	carry
neskoková	00	1	x	x
JMP	01	0	x	x
JMZ	01	0	1	x
JMC	01	0	x	1
RET	10	1	x	x

Legenda: x ... libovolná hodnota

tab. 4.1 Nastavení řídících signálů pro programový čítač

V tomto projektu je implementován rozšířený programový čítač, blokové schéma na obr. 4.5, jenž umí pracovat s nepodmíněnými a podmíněnými skoky a návraty z podprogramů.

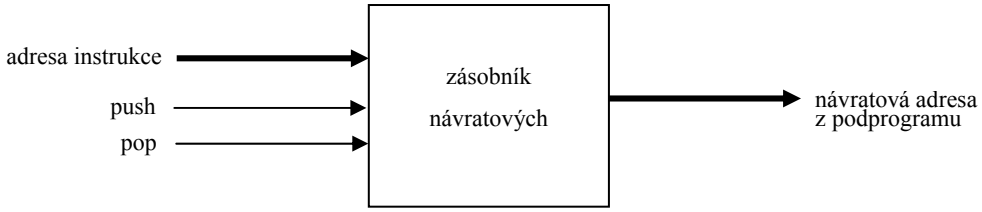


obr. 4.5 Blokové schéma programového čítače

4.5.2 Zásobník návratových adres

Pro podporu instrukcí volání a návratů z podprogramů slouží v procesoru tato jednotka. Tady jsou uchovávány adresy, ze kterých se do podprogramů skočilo, aby po ukončení podprogramu mohl program dále pokračovat na původním místě.

Na základě instrukce volání podprogramu CALL, je nastaven řídicí signál push, do zásobníku je uložena stávající adresa instrukce a inkrementuje se hodnota čítače, který slouží jako ukazatel na vrchol zásobníku. Při provádění instrukce návratu z podprogramu RET se nastává signál pop, vyzvedne se adresa z vrcholu zásobníku, pošle se do programového čítače a provede se dekrementace čítače. Hloubka zásobníku určuje, kolik současných zanoření do podprogramů je možné provést. Blokové schéma zásobníku návratových adres je zobrazeno na obr 4.6.



obr. 4.6 Blokové schéma zásobníku návratových adres

4.5.3 Aritmeticko-logická jednotka

Je ústředním výpočetním prvkem procesoru. Odehrávají se zde, všechny aritmetické a logické operace, které procesor podporuje.

K uskutečnění aritmetických operací potřebujeme sčítačku a odečítačku. K odečítání ale můžeme použít také sčítačku v případě, že záporné číslo nahradíme jeho dvojkovým doplňkem. Potom nám pro sčítání i odečítání bude stačit pouze sčítačka. Dvojkový doplněk se realizuje negací operandu a přičtením jedničky. Protože náš instrukční soubor obsahuje ještě další operace sčítání a odečítání se vstupním přenosem, budeme definovat operace sčítání a odečítání operandů A a B, kde C bude označovat vstupní přenos C, takto:

sčítání bez přenosu: A + B

sčítání s přenosem: A + B + C

odečítání bez přenosu: A – B = A + negace (B) + 1

odečítání s přenosem: A – B – C = A + negace (B) + negace (C)

Vstupní přenos bude vstupovat do ALU jako signál, ale bude využíván jen v případě aritmetických operací a u operace posunu jako nasouvaný bit. Operace odečítání bez přenosu je využívána také k realizaci operace porovnání dvou registrů. Z tab. 4.2 je zřejmé, které řídící signály ovlivňují chování jednotky při aritmetických operacích a jakým způsobem.

instrukce	řídicí signály			
	oper_group	arith_oper	incl_carry	cin
bez přenosu				
ADD	00	0	0	x
SUB	00	1	0	x
s přenosem				
ADD	00	0	1	C
SUB	00	1	1	C

Legenda: x ... libovolná hodnota

C ... vstupní přenos

tab. 4.2 Hodnoty řídících signálů pro ALU při vykonávání aritmetických operací

Pro logické operace využívám v aritmeticko-logické jednotce logické operace jazyka VHDL. Nastavení signálů pro logické operace a operaci přesunu mezi registry je v tab 4.3.

instrukce	řídicí signály	
	oper_group	logic_oper
AND	01	00
OR	01	01
XOR	01	10
MOV	01	11

tab. 4.3 Hodnoty řídicích signálů pro ALU při vykonávání logických operací

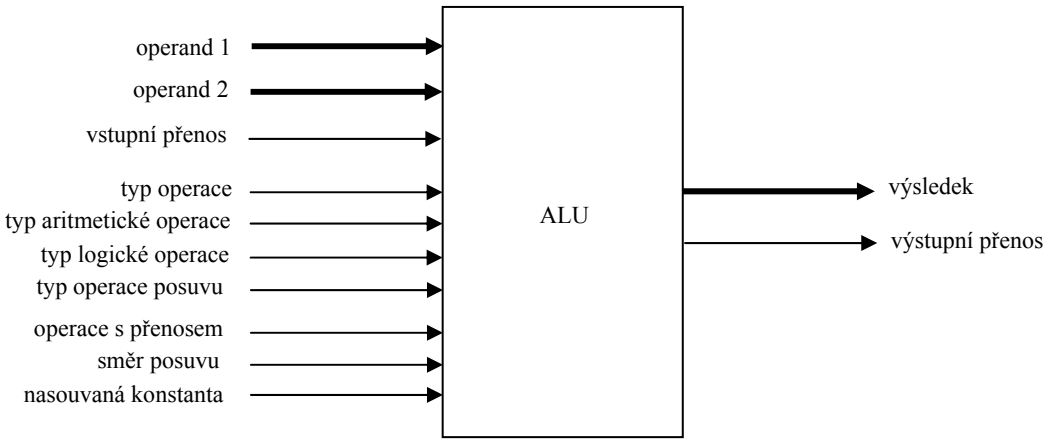
Jednotka obsluhuje také operace logických posuvů a rotací. Vykonání operace závisí na typu instrukce a je řízeno nastavením odpovídajících signálu. Tab 4.4 ukazuje typy instrukcí logických posuvů a rotací, které umí navržená aritmeticko-logická jednotka provádět, a jim příslušející řídicí signály.

instrukce	řídicí signály					nasouvaný bit
	oper_group	shift_oper	shift_dir	shift_const	cin	
SL0	10	00	0	0	x	shift_const
SL1	10	00	0	1	x	shift_const
SLA	10	01	0	x	C	cin
SLX	10	10	0	x	x	LSB
ROL	10	11	0	x	x	MSB
SR0	10	00	1	0	x	shift_const
SR1	10	00	1	1	x	shift_const
SRA	10	01	1	x	C	cin
SRX	10	10	1	x	x	MSB
ROR	10	11	1	x	x	LSB

Legenda: x ... libovolná hodnota
C ... vstupní přenos
LSB ... nejnižší bit
MSB ... nejvyšší bit

tab. 4.4 Hodnoty řídicích signálů pro ALU při vykonávání operací posuvů a rotací

Pro rozhodnutí o uskutečnění té či oné operace vstupují do aritmeticko-logické jednotky dvě množiny signálů, z nichž jedna určuje typ operace, která se bude provádět, a druhá určuje, s jakými parametry se operace bude provádět. Blokové schéma aritmeticko-logické jednotky můžeme vidět na obr. 4.7.



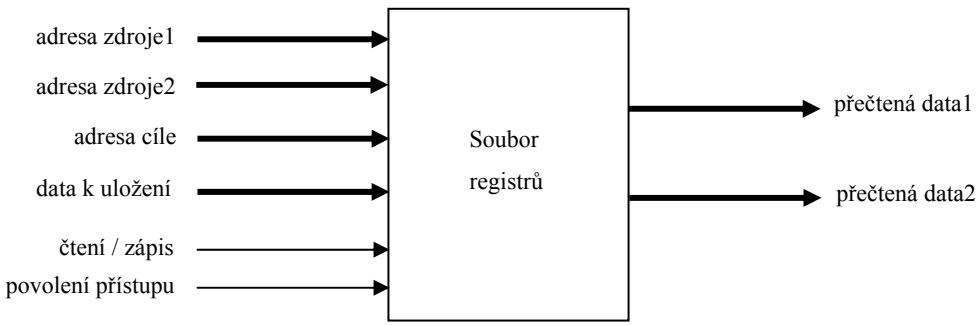
obr. 4.7 Blokové schéma aritmeticko-logické jednotky

4.5.4 Instrukční dekodér

Činností instrukčního dekodéru je z instrukčního kódu nahraného v instrukčním registru vydekódovat informace potřebné k vykonání příslušné instrukce. Instrukční kód je rozebrán na základní elementy a je provedena transformace operačního kódu instrukce na hodnoty všech signálů, kterými lze ovlivnit činnost částí procesoru potřebných k úspěšnému provedení instrukce.

4.5.5 Registry

Sada registrů je speciálnější případ paměti. Ke své činnosti potřebuje dva vstupy pro čtení a jeden pro zápis, protože máme dva zdrojové operandy a jeden cílový. Pro zápis nebo čtení z paměti musí být nastaven signál povolení přístupu k registru a nastaven signál čtení/zápis na příslušnou hodnotu. Při čtení z registrů musí být nastaveny identifikace registrů, ze kterých požadujeme čtení, a na výstupu se potom objeví příslušné přečtené hodnoty. Při zápisu do registru musíme nastavit identifikaci registru, do kterého chceme zapisovat a vložit na vstup data, která chceme zapsat. Blokové schéma souboru registrů je na obr. 4.8.



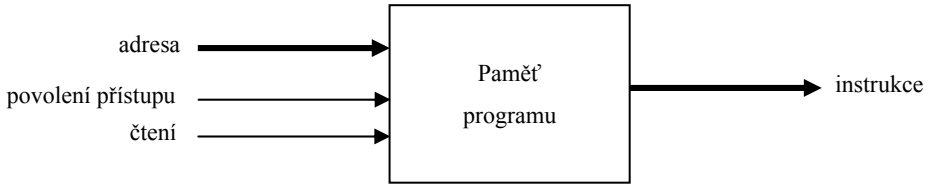
obr. 4.8 Blokové schéma souboru registrů

4.5.6 Paměti

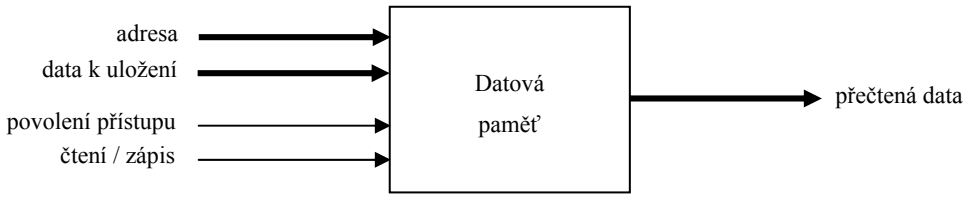
Paměti jsou nezbytnou součástí navrhovaného procesoru. Jedná se o paměť programu a paměť dat ve struktuře podle Harvardské koncepce. Schémata obou pamětí jsou vyobrazena na obr. 4.9 a obr. 4.10.

V paměti programu jsou uchována data, která řídí program. Musí být naplněna před samotným spuštěním procesoru, protože v průběhu práce procesoru již není možné do této paměti zapisovat, pouze ji číst. Pokud do paměti nezapíšeme žádná data může se stát, že v paměti zůstanou data náhodná a potom při spuštění procesoru může dojít k neočekávanému chování. Pro přečtení instrukce z programové paměti musí být na adresovou sběrnici vystavena adresa a musí byt nastaven signál pro povolení přístupu a čtení.

Datová paměť slouží k uložení dat, se kterými program pracuje. Tato paměť narozdíl od výše uvedené programové paměti umožňuje jak zápis, tak čtení, ale pouze při vykonávání programu. Operace zápisu a čtení jsou řízeny povolovacím signálem pro přístup do paměti a signálem určujícím, zda požadujeme čtení nebo zápis. Při čtení musí být na adresovou sběrnici vystavena adresa buňky, ze které chceme číst a na datové sběrnici se objeví požadovaná data. Operace zápisu požaduje vystavit adresu na adresovou sběrnici a data, která mají být zapsána na sběrnici datovou.



obr. 4.9 Blokové schéma instrukční paměti



obr. 4.10 Blokové schéma datové paměti

4.5.7 Implementace

Procesor byl sestaven propojením všech výše navržených komponent, tak aby fungoval následovně. Programový čítač generuje adresu pro paměť programu. Instrukce je nahrána z paměti programu do instrukčního registru. Dekodér tuto instrukci rozdekóduje na jednotlivé bity, které použije řadič pro řízení komponent, které jsou potřeba pro správné vykonání aktuální instrukce. Podle dané instrukce nastavuje signály registrům, aritmeticko-logické jednotce, je jimi řízena datová paměť, nebo v případě instrukcí skokových také programový čítač.

5 Návrh paměti cache

Prvním krokem ke zvýšení výkonu stávajícího procesoru, jehož návrhu a implementaci byla věnována předcházející kapitola 4, bude modifikace jeho paměťového systému přidáním rychlé vyrovnávací paměti.

V úvodní části této kapitoly vysvětlím, proč jsem se rozhodla navržený procesor obohatit právě o paměti cache. V další části si popíšeme, jaké bude nutné provést změny v již navržených pamětech pro data a instrukce a nakonec bude následovat samotný návrh a popis implementace pamětí cache pro data a instrukce.

5.1 Výhody a nevýhody paměti cache

Principům pamětí cache je věnována kapitola 3.1, a proto zde provedu jen stručné shrnutí základních informací o těchto prvcích.

Paměti cache jsou vyráběny jako rychlé paměti a slouží k vyrovnání rychlostí mezi rychlým procesorem a pomalou operační pamětí. V případě, že procesor nepoužívá tuto vyrovnávací paměť, pak je jeho činnost přístupy do operační paměti velmi zpomalována. Kvůli požadavkům na vysokou rychlost paměti cache je však omezena její kapacita, která je mnohem menší než kapacita operační paměti. Následkem toho dochází k výpadkům, kdy data požadovaná procesorem nejsou přítomna v paměti cache, a pak nezbývá než je zavést z operační paměti.

Činnost paměti cache vychází z toho, že program zpracovávaný procesorem se většinou při své práci zdržuje určitou dobu na určitém místě paměti, což platí při načítání instrukcí i při zpracování dat ve spolupráci s pamětí. Pokud procesor požaduje nějakou informaci z paměti, hledá se nejprve v příslušné paměti cache. Teprve v případě, kdy zde tato informace není nalezena, nastane spolupráce s hlavní pamětí a dojde k zavedení těchto dat z hlavní paměti do paměti cache. Pokud dojde k zaplnění paměti cache a je potřeba zavést další data, musí se pomocí zvoleného algoritmu potřebná část paměti uvolnit. Princip výměny dat mezi hlavní pamětí a pamětí cache je popsán v kapitole 3.1.3, kde jsou kromě některých algoritmů, které se používají při uvolňování paměti cache, vysvětleny také způsoby zpětného zápisu dat modifikovaných procesorem z paměti cache do hlavní paměti.

5.2 Modifikace operační paměti

V procesoru popsaném v kapitole 4 byla operační paměť rozdělena podle Harvardské koncepce na dvě samostatné paměti, jednu pro data a druhou pro instrukce. Toto pojetí nebudeme měnit a nadále tedy zůstane zachováno.

Základní vlastnosti obou pamětí nebudou přidáním paměti cache také žádným způsobem ovlivněny. Instrukční paměť bude nadále uchovávat program v podobě jednotlivých instrukcí, jehož provedení bude zajištěno zpracováním instrukcí procesorem, a zachová si také vlastnost, že do ní není možné zapisovat, ale je určena pouze pro čtení. Datová paměť bude dále sloužit k ukládání dat, se kterými procesor pracuje, a ponechává možnost procesoru při zpracovávání programu odsud data číst a také je sem zapisovat.

Vložení paměti cache do paměťové hierarchie počítače tedy přinese změny především v komunikaci uvnitř této paměťové hierarchie s tím, že pomineme vnitřní registry procesoru. Paměť cache bude umístěna mezi operační paměť a procesor, a proto musíme nově vytvořit komunikační spojení mezi pamětí cache a hlavní pamětí a mezi pamětí cache a procesorem. Ze strany procesoru zůstane postup při čtení dat z paměti a zápisu dat zpět do paměti téměř stejný, akorát nyní nebude komunikovat přímo z operační pamětí, nýbrž s pamětí cache a jejím řadičem. Největší změny budou tedy provedeny na straně hlavní paměti, protože komunikace mezi touto pamětí a pamětí cache bude probíhat na jiné úrovni než probíhala dříve s procesorem. Mezi hlavní pamětí a procesorem byly vždy přenášeny jen jednotlivé informace, ale nyní budou mezi touto pamětí a pamětí cache přenášeny celé bloky dat. Celá hlavní paměť bude tedy rozdělena na stejně velké bloky dat, jejichž velikost bude určovat kapacitu dat, které je možné uložit na jednom řádku v paměti cache. U rozhraní hlavní paměti tak bude nutné přizpůsobit šířku datových vstupů a výstupů na velikost jednoho bloku dat.

5.3 Návrh a implementace pamětí cache

Vzhledem k tomu, že jsem se při návrhu pamětí procesoru rozhodla jít cestou rozdělení operační paměti pro instrukce a data, budu se nyní zabývat návrhem oddělených pamětí cache pro data a instrukce. Oddělení pamětí cache přináší při návrhu a implementaci podobné výhody jako při návrhu oddělených hlavních pamětí. Kromě možnosti zvolit různé kapacity pamětí, se tady nabízí možnost volby dalších parametrů, jako způsobu mapování, algoritmu pro výběr bloku při uvolňování místa v paměti nebo strategie zápisu modifikovaných dat do hlavní paměti, která se ale u instrukční cache paměti neuplatňuje.

Mapování bloku dat do paměti cache lze provést několika různými způsoby, které byly podrobně rozebrány v kapitole 3.1.2. U paměti cache s přímým mapování dochází k ukládání určitého bloku vždy na stejný řádek cache tabulky, proto nám při hledání stačí jen podle adresy určit odpovídající řádek a porovnat, zda tag hledaného bloku odpovídá tagu uloženému na tomto řádku. Nevýhoda tohoto způsobu mapování spočívá v častých výpadcích, které nastávají v situaci, kdy chceme opakovaně pracovat s bloky, jež se mapují do stejného řádku cache tabulky. Tuto nevýhodu lze eliminovat použitím plně asociativního mapování, u kterého může být blok uložen do libovolného řádku cache tabulky. Hledání bloku v takové paměti pak probíhá tak, že se porovnává tag hledaného bloku s tagem na každém řádku tabulky cache. Kvůli vyhledávání je tedy potřeba použít velké

množství komparátorů, a proto se konstruují asociativní paměti jen s menší kapacitou. Výhody obou předchozích způsobů spojuje n-cestně asociativní mapování, kde se při hledání bloku nejprve vybere řádek přímým způsobem, a potom se asociativním způsobem porovnává vstupní tag s tagy uloženými na tomto řádku ve všech n tabulkách. Nejjednodušší z hlediska konstrukce je dvoucestně asociativní cache, která má oproti ostatním n-cestně asociativním cache jednodušší logiku, ale také nižší úspěšnost hledání. Obecně lze říci, že s počtem cest roste úspěšnost hledání, ale také náklady na konstrukci.

V tomto projektu jsem pro obě paměti vybrala nejpoužívanější n-cestně asociativní způsob mapování a počet cest rovný dvěma. Implementace tohoto způsobu mapování je po přímém mapování nejjednodušší a přitom vykazuje lepší výsledky při hledání než přímé mapování.

5.3.1 Paměť cache pro data

Tato paměť je umístěna mezi stávající operační datovou paměť a procesor a umožňuje jak čtení, tak zápis dat. Její činnost spočívá v uchovávání bloků dat přenesených sem z hlavní datové paměti, aby mohla být tato data využívána procesorem při zpracovávání programu, a také v zajištění zpětného zápisu bloků dat modifikovaných procesorem zpět do hlavní datové paměti a to na základě principu opožděného zápisu (write-back).

Navržená paměť je dvoucestně asociativní, což znamená, že se skládá ze dvou tabulek. Každá tabulka pak obsahuje 8 řádků a je rozdělena na několik sloupců. V jednom sloupci jsou umístěny tagy, tedy klíče, podle kterých se bloky v paměti asociativně vyhledávají. Další dva sloupce uchovávají samotné bloky dat tak, že každý sloupec je určen pro jedno slovo bloku. Blok přenášený mezi hlavní pamětí a pamětí cache tedy sestává ze dvou slov, což jsou vlastně dva řádky v hlavní paměti. Zbývající dva sloupce každé tabulky jsou určeny pro uložení pomocných informací, které jsou nutné ke správné funkci paměti. Jedná se o informace o platnosti uložených dat a informace o modifikaci uložených dat ze strany procesoru. Dále paměť cache obsahuje jeden sloupec, který je společný pro obě tabulky a obsahuje informace o tom, do které z tabulek se přistoupilo naposledy, tedy informaci potřebnou pro realizaci algoritmu LRU (Least Recently Used Strategy). Výsledný vzhled celé paměti cache je zobrazen na obr. 5.1.

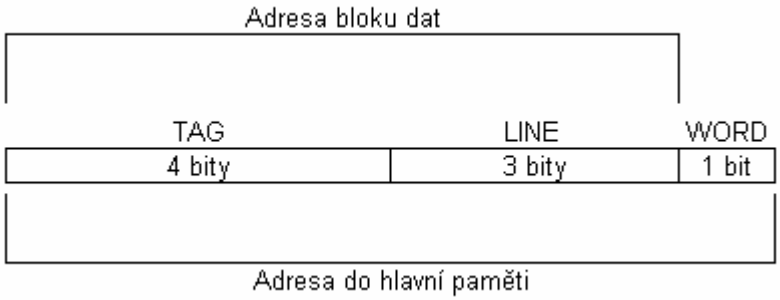
Abychom mohli v této paměti vyhledávat, musíme znát adresu hledaného datového slova. Tato adresa je vystavena procesorem, pokud požaduje číst data z paměti nebo zapisovat data do paměti, a řadičem paměti cache je rozdělena na části, které jsou přivedeny na odpovídající vstupy paměti cache. Rozdělení adresy je zobrazeno na obr 5.2. V našem případě má adresa přivedená z procesoru velikost 8 bitů a sestává ze tří částí. Nejnižší bity identifikují slovo uvnitř bloku, v našem případě je to 1 bit, protože blok obsahuje jen dvě slova, každé o velikosti 8 bitů. Další bity tvoří adresu řádku v tabulkách paměti cache. Vzhledem k tomu, že každá tabulka obsahuje sedm řádků, tvoří tuto část

adresy 3 bity. Zbývající 4 bity jsou považovány za tag a společně se třemi bity adresy řádku jednoznačně identifikují každý blok hlavní paměti.

dvoucestná paměť cache

	tabulka 1						tabulka 2					
bit LRU	adresa řádku	bit platnosti	bit modifikace	TAG	WORD 0	WORD 1	adresa řádku	bit platnosti	bit modifikace	TAG	WORD 0	WORD 1
	0						0					
	1						1					
	2						2					
	3						3					
	4						4					
	5						5					
	6						6					
	7						7					

obr. 5.1 Schéma navržené 2-cestné paměti cache

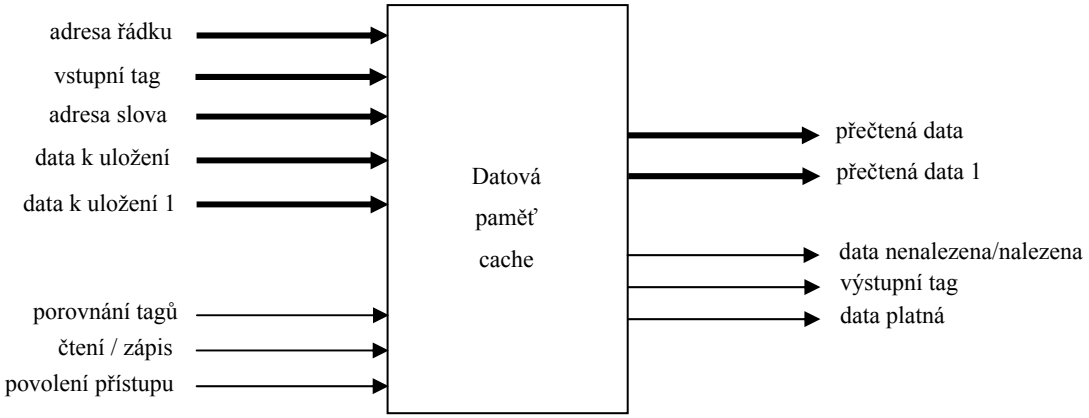


obr. 5.2 Rozdělní adresy do hlavní paměti pro navrženou paměť cache

Datová paměť cache může pracovat ve čtyřech režimech. Konkrétní režim je zvolen na základě vstupních řídicích signálů nastavených řadičem paměti cache. Celkové blokové schéma této paměti je na obr. 5.3. Všechny činnosti v jednotlivých režimech lze provádět jen v případě, že je činnost paměti aktivována.

V režimu čtení dat procesorem, tedy pokud procesor zpracovává instrukci load, musí procesor vystavit adresu dat, která chce přečíst. Tuto adresu řadič rozdělí na jednotlivé části, ty pošle do paměti cache a paměť cache odpoví, zda data byla nalezena (cache hit) či nikoliv (cache miss). V případě, že data nalezena byla, jsou tato data ihned předána do procesoru. V opačném případě paměť cache řadiči sdělí informaci o tom, zda jsou v obou tabulkách na daném řádku uložena platná data a pokud jsou, tak vybere tabulku podle algoritmu LRU a sdělí o těchto datech informaci, zda byla modifikována. Řadič paměti cache vyšle požadavek na přečtení hledaných data do hlavní paměti. Pokud v paměti cache není pro tyto data místo a zároveň na řádku vybraném algoritmem LRU jsou uložena modifikovaná data, tak nastaví paměť cache do režimu přečtení celého bloku dat. Poté co je přečten

požadovaný blok dat z hlavní paměti, tak je paměť cache nastavena do režimu zápisu bloku dat a tento blok je sem zapsán. V případě, že byl z paměti cache načten blok modifikovaných dat, vyšle zároveň řadič požadavek do hlavní paměti na zápis tohoto bloku dat. Až se dokončí všechny tyto operace jsou požadovaná data předána procesoru.



obr. 5.3 Blokové schéma datové paměti cache

Při požadavku na zápis dat ze strany procesoru, tedy když vykonává instrukci store, nastaví procesor adresu, kam chce zapisovat, a data k zapsání. Řadič opět tuto adresu zpracuje, pošle ji do paměti cache, zároveň sem pošle také daná data a nastaví režim zápisu dat. Pokud jsou na daném řádku nalezena platná původní data, jsou tato nahrazena novými a je nastaven příznak, že data byla modifikována. V případě, že původní data v paměti cache nalezena nebyla nebo nejsou platná, informuje paměť cache řadič, jestli je daný řádek plně obsazen platnými daty a pokud je, tak podle algoritmu LRU zjistí o příslušných datech, zda byla modifikována. Řadič paměti cache vyšle požadavek do hlavní paměti, aby odsud byl přečten blok dat, do kterého mají být zapsána data z procesoru, a pokud je potřeba uvolnit místo pro tento nový blok v paměti cache a data zde uložená byla modifikovaná, vyšle řadič do paměti cache požadavek na přečtení tohoto modifikovaného bloku. Po přečtení bloku dat z hlavní paměti, je tento uložen do paměti cache a modifikovaný blok je zapsán zpět do hlavní paměti. Nakonec jsou zapsána do paměti cache data vystavená procesorem.

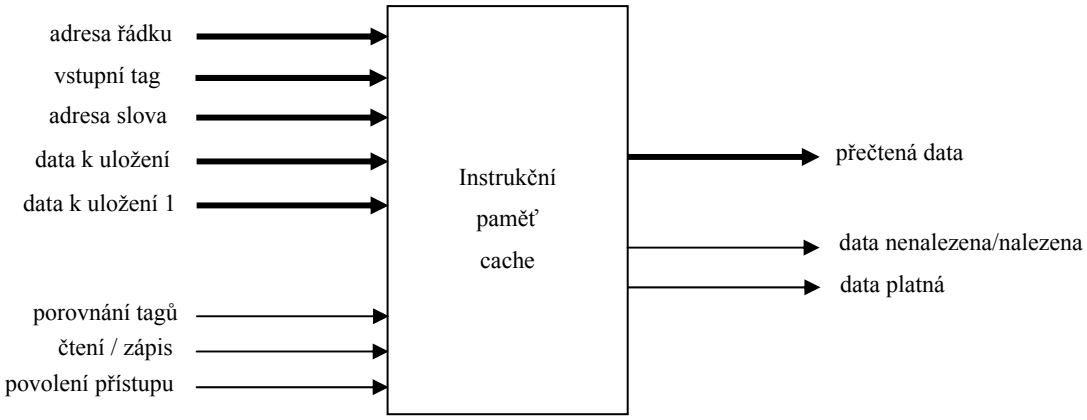
5.3.2 Paměť cache pro instrukce

Instrukce jsou uloženy v operační paměti pro instrukce a původně odsud byly přímo načítány do procesoru. Tento způsob načítání byl pomalý, proto byla mezi hlavní paměť instrukcí a procesor vložena instrukční paměť cache. Tato paměť je ze strany procesoru omezena pouze na čtení a ze strany hlavní paměti pouze na zápis. Jejím úkolem je uchovávat bloky instrukcí, aby mohly být pak jednotlivé instrukce procesorem rychle načítány.

Architektura této paměti cache ani způsob adresace se neliší od datové paměti cache a jejich podrobným popisem se zabývám v kapitole 5.3.1. Jediný rozdíl v architektuře oproti datové paměti

cache spočívá v tom, že tabulky této paměti cache neobsahují sloupec, který nese informaci o modifikaci uložených dat. Princip adresace je úplně stejný jako u datové paměti cache, pouze dochází ke změně velikosti slova bloku na 16 bitů, což je počet bitů, které zabírá jedna instrukce.

Tato paměť pracuje pouze ve dvou režimech, které jsou opět voleny na základě vstupních signálů, a její blokové schéma je na obr. 5.4.



obr. 5.4 Blokové schéma instrukční paměti cache

Pokud procesor potřebuje přečíst instrukci, tedy ve fázi výběru instrukce z paměti, musí nastavit její adresu. Tuto adresu zpracuje řadič instrukční paměti cache a přivede ji na vstup instrukční paměti cache a zároveň jí nastaví režim čtení dat. Pokud se požadovaná instrukce bude nacházet v této paměti (cache hit), bude ihned předána procesoru. Jinak (cache miss) musí řadič paměti cache vyslat požadavek do hlavní paměti, aby zde byl přečten celý blok s požadovanou instrukcí. Až bude mít tento blok k dispozici, nastaví paměť cache do režimu zápisu bloku dat a tento blok jí předá. Poté je požadovaná instrukce předána procesoru.

Práce řadiče instrukční paměti cache je mnohem jednodušší než práce řadiče datové cache, protože se nemusí zabývat zápisem dat do paměti cache ze strany procesoru a při zápisu bloku dat ze strany hlavní paměti tak pouze vybere místo, které je volné nebo podle algoritmu LRU, a data sem zapíše.

6 Návrh zřetězeného zpracování instrukcí

Náplní této kapitoly bude návrh zřetězeného zpracování instrukcí. Tato změna způsobu zpracování instrukcí by měla vést k dalšímu zvýšení výkonu původního procesoru popisovaného v kapitole 4, který byl kvůli požadavkům na lepší výkonnost dále obohacen přidáním pamětí cache, jejichž návrhu byla věnována předcházející kapitola 5.

Na začátku bude provedeno shrnutí základních informací, které se vztahují k zřetězenému zpracování. Toto bude zahrnovat stručný popis přínosů tohoto principu oproti sekvenčnímu zpracování, ale zmíním se zde také o problémech, které s sebou tento způsob zpracování přináší.

V rámci samotného návrhu, kterému bude věnována zbývající část této kapitoly, se nejprve budeme zabývat výběrem konkrétní linky pro zřetězené zpracování instrukcí s ohledem na instrukční soubor. Dále již bude následovat samotný návrh vybrané linky. Nejdříve bude navržena jednoduchá linka, která nebude potlačovat konflikty vznikající při zpracovávání, a ta bude poté vylepšována implementací technik, jejichž úkolem bude tyto konflikty eliminovat.

6.1 Klady a zápory zřetězeného zpracování instrukcí

Podrobně byla problematika zřetězeného zpracování instrukcí rozebrána v kapitole 3.2, proto zde provedu jen shrnutí nejdůležitějších poznatků a detaily se zde zabývat nebudu.

Zřetězené zpracování instrukcí (pipelining) lze chápat jako techniku, při které dochází k překrytí vykonávání několika instrukcí. Vychází při tom z toho, že vykonání instrukce lze rozdělit na určitý počet na sobě nezávislých akcí, které můžou být pro různé instrukce vykonávány paralelně. U sekvenčního zpracování lze sice vykonání instrukce také rozdělit na několik částí, které ale nebývají na sobě nezávislé, a jeho princip tedy spočívá v tom, že zpracování další instrukce může začít až poté, kdy je úplně dokončeno vykonání předchozí rozpracované instrukce.

Pokud budeme uvažovat případ, že na vstupu bude k dispozici vždy dostatečný počet instrukcí čekajících na zpracování, potom budou výsledky zpracování jednotlivých instrukcí na výstupu u sekvenčního zpracování vždy v časových intervalech daných dobou průchodu instrukce celou linkou pro zpracování. U zřetězeného zpracování bude sice první výsledek dostupný na výstupu za dobu stejnou jako u sekvenčního zpracování, ale následující výsledky se budou objevovat na výstupu v intervalech daných periodou hodin, která je určena dobou trvání nejpomalejšího stupně linky a zpožděním záchytného registru.

Situace, kdy se při zřetězeném zpracování objevují výsledky na výstupu pravidelně v intervalech daných periodou hodin procesoru, nastává pouze v ideálním případě, pokud jsou instrukce zpracovávaného programu na sobě nezávislé. Tato situace je ovšem v praxi nedosažitelná, protože instrukce jsou na sobě často závislé, což může vést k tomu, že určitá instrukce nemůže být provedena dříve, než se dokončí zpracování jedné nebo více předchozích instrukcí. Závislosti mezi instrukcemi můžeme rozlišit na několik typů, z nichž některé je možné úplně odstranit, ale jiné lze pouze eliminovat. Závislosti, které není možné úplně odstranit, pak vedou ke vzniku konfliktů při zřetězeném zpracování a zastavování linky při zpracovávání.

U sekvenčního zpracování sice k žádným konfliktům nedochází, protože každá instrukce má všechny výsledky předchozích instrukcí k dispozici již v době, kdy začíná její zpracování, ale čekání na dokončení předchozí rozpracované instrukce je příliš dlouhé a většina výpočetních prostředků zůstává při čekání nevyužita. Při zřetězené zpracování je snaha využívat celou linku v každém taktu v maximální možné míře tím způsobem, že v každém stupni linky pracují ty prostředky, které odpovídají potřebám právě zpracovávané instrukce.

6.2 Výběr instrukční linky

Před samotným návrhem a implementací zřetězeného zpracování instrukcí do procesoru, který dosud zpracovával instrukce programu sekvenčním způsobem, bude nezbytné vybrat vhodný způsob zřetězeného zpracování jednotlivých skupin instrukcí instrukčního souboru uvažovaného procesoru.

Dále se budeme zabývat návrhem zřetězeného zpracování instrukcí pro procesor, který byl popsaný v kapitole 4 a který lze charakterizovat několik základními vlastnostmi. Všechny operace s daty provádí pouze nad daty uloženými v registrech. Přístup do paměti umožňuje jen prostřednictvím instrukcí přesunů (load/store), které zajišťují přesun dat mezi pamětí a registry. Instrukce mají pevnou délku 16 bitů a lze je rozdělit podle funkce do tří skupin.

Aritmeticko-logické instrukce pracují se dvěma operandy uloženými v registrech, nebo jedním operandem v registru a druhým tvořeným konstantou, která je součástí instrukce. Vlastní operace nad těmito operandy je provedena aritmeticko-logickou jednotkou a výsledek operace je vždy ukládán do registru.

Další skupinu tvoří instrukce přesunů mezi datovou pamětí a registry, které mohou být z hlediska formátu instrukce stejných typů jako aritmeticko-logické instrukce. U instrukce ukládání dat do paměti jsou přenášená data vždy uložena v registru a úplná adresa do datové paměti je v závislosti na formátu buď uložena v registru, nebo je součástí instrukce. V případě instrukce nahrávání dat z paměti slouží vždy registr jako cíl vybraných dat a zdrojem úplné adresy do paměti může být opět registr nebo samotná instrukce. Pro úspěšné vykonání operací přesunů je třeba umožnit přístup k registrům a do datové paměti.

Poslední skupinou jsou instrukce skokové, které na základě splnění dané podmínky nebo nepodmíněně mění řízení běhu programu. Tyto instrukce v sobě nesou zakódovanou úplnou adresu cílové instrukce skoku, která je po dekódovaní instrukce a případně ověření podmínky skoku předána programovému čítači. Pro zpracování těchto instrukcí tedy nepotřebujeme žádné další výpočetní prostředky.

Na základě zjištěných poznatků o tom, jaké prostředky procesor potřebuje při zpracovávání jednotlivých skupin instrukcí, jsem se rozhodla pro návrh linky pro zřetězené zpracování instrukcí se čtyřmi stupni, která je typická pro tradiční procesory RISC.

6.3 Návrh a implementace 4-stupňové linky

Při návrhu začneme tím, že si nejprve rozdělíme původní sekvenční zpracování do několika částí, ve kterých se budou vykonávat navzájem nezávislé činnosti. Toto rozdělení se pak stane základem linky pro zřetězené zpracování instrukcí.

Na základě rozboru provedeného v předchozím oddílu 6.2 jsem zjistila, že k provedení každé instrukce uvažovaného procesoru jsou potřeba maximálně 4 takty. Zřetězené zpracování sice dokáže zvýšit počet instrukcí zpracovaných procesorem za jednotku času, avšak nedokáže zkrátit dobu zpracování jedné instrukce, proto zůstane doba zpracování každé instrukce stejná jako u sekvenčního zpracování. Úkolem tedy je rozdělit činnost procesoru potřebnou ke zpracování instrukce do čtyřech na sobě nezávislých fází tak, aby v jednom taktu mohla být v každé fázi zpracování jiná instrukce. K označení jednotlivých fází jsem převzala zkratky IF (Instruction Fetch), ID (Instruction Decode), EX (Instruction Execute) a WB (Write Back) a činnost v jednotlivých fázích je následující.

Fáze IF:

V této fázi se nejprve aktualizuje čítač instrukcí, který poté pošle adresu do paměti instrukcí a z paměti se podle této adresy vybere příslušná instrukce.

Fáze ID:

Instrukce, která byla v předcházejícím taktu načtena z paměti instrukcí, je v této fázi dekódována, zároveň dochází k načtení operandů z registrů, jejichž adresy byly součástí instrukce, a také se z instrukce vybere konstanta. Čtení z registrů a výběr konstanty mohou probíhat paralelně s dekódováním instrukce, protože identifikátory registrů i konstanta mají v instrukčním kódu své pevné místo. Současně probíhá vyhodnocení obsahu podmínkových registrů pro případ podmíněných skoků a na konci této fáze se, v případě instrukce skoku a splnění podmínky skoku, do programového čítače posílá cílová adresa skoku. V případě aritmeticko-logických instrukcí a instrukcí přesunů se po dekódování ještě určí, zda druhý operand bude konstanta nebo obsah registru.

Fáze EX:

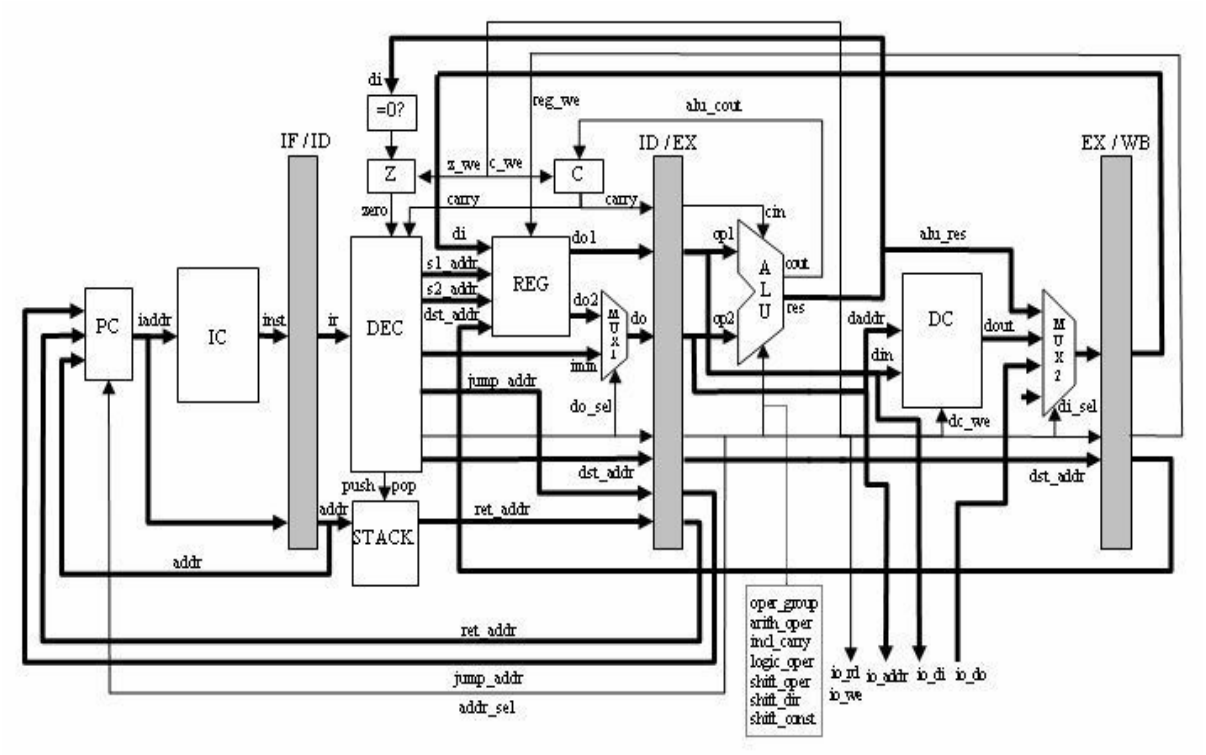
Operandy, které byly připraveny v předchozí fázi jsou přivedeny na vstupy aritmeticko-logické jednotky a ta provede operaci v závislosti na typu instrukce. Vzhledem k tomu, že součástí instrukce je už kompletní adresa do paměti a není tedy potřeba v této fázi adresu dopočítávat, přesuneme do této fáze také operace, které by jinak byly provedeny ve fázi MA (Memory Access) a tuto fázi zrušíme. Pokud se jedná o instrukci load, přečtou se z datové paměti data uložená na adrese, která je tvořena obsahem druhého operandu. V případě instrukce store se do datové paměti uloží data, která byla přečtena ve fázi dekódování z registru a uložena do prvního operandu, a obsah druhého operandu budeme chápat jako adresu do datové paměti.

Fáze WB:

Do této fáze se dostáváme jen v případě, pokud se zpracovává některá instrukce ze skupiny aritmeticko-logických instrukcí nebo pokud jsme v předchozí fázi prováděli čtení datové paměti pomocí instrukce load. Výsledek vypočítaný aritmeticko-logickou jednotkou nebo data přečtená z paměti, v závislosti na typu instrukce, jsou uloženy zpět do registru, jehož identifikace byla součástí kódu instrukce.

Při této implementaci instrukční linky trvá zpracování skokových instrukcí 2 takty, aritmeticko-logických instrukcí a instrukce čtení z paměti 4 takty a instrukce zápisu do paměti 3 takty.

Zřetězené zpracování instrukcí vytvoříme tak, že převezmeme beze změn stávající navrženou linku se čtyřmi stupni zpracování a v každém hodinovém taktu začneme zpracovávat novou instrukci. Protože při zřetězeném zpracování jsou všechny stupně linky aktivní v každém cyklu, musí být splněna podmínka, že všechny operace prováděné v jednom stupni budou vždy dokončeny v jednom taktu. Nejdůležitější je ovšem přidat mezi jednotlivé stupně oddělovací tzv. záchytné registry, aby sem mohly být ukládány mezivýsledky vyprodukované jednotlivými stupni při přechodu instrukce z jednoho stupně do druhého. Pokud bychom tyto registry do linky pro zřetězené zpracování nepřidali, mohou být hodnoty vyprodukované v některém z předchozích stupňů přepsány hodnotami novými dříve, než je stihne zpracovat stupeň, pro který byly určeny. Příkladem může být předávání identifikace cílového registru, když je zpracovávána instrukce, která využívá ALU, nebo instrukce čtení z paměti, kdy se tento údaj vytvořený ve stupni ID předává beze změny přes stupeň EX až do stupně WB, kde se využívá při ukládání výsledku vypočteného aritmeticko-logickou jednotkou nebo dat přečtených z paměti. Na obr. 6.1 je zobrazené schéma procesoru se základní 4-stupňovou linkou pro zřetězené zpracování instrukcí bez řešení závislostí mezi instrukcemi. Jednotlivé záchytné registry jsem pojmenovala podle stupňů, mezi kterými se nachází. Závislosti, které mezi instrukcemi při zpracování vznikají, jsou zatím odstraňovány při kompilaci programu a jejich řešením úpravou linky se budeme zabývat dále v oddílu 6.4.



obr. 6.1 Schéma navrženého procesoru se základní 4-stupňovou linkou

Operace, které se provádějí v fázi IF a ID, jsou nezávislé na typu právě zpracovávané instrukce, protože v této době ještě nebylo provedeno nebo dokončeno dekódování a není tedy známo, o jaký typ instrukce se jedná. Ve fázi ID se po dekódování instrukce provede v závislosti na typu instrukce nastavení řídicích signálů pro všechny řídicí i výpočetní prostředky. Ve fázi ID je řízeno nastavení druhého operandu, kde vybíráme mezi daty přečtenými z registru a konstantou. V případě, že se v ID zjistí instrukce skoku a je splněna podmínka skoku, pak se z fáze ID ovlivňuje také chování programového čítače v příští fázi IF. Řídicí signály vygenerované ve fázi ID potřebné ve fázi EX a WB jsou uloženy do záchytného registru ID/EX. Ve fázi EX se přivedou signály na vstupy ALU, datové paměti a vstupněvýstupní porty a podle toho, jak byly tyto signály ve fázi ID nastaveny se provedou příslušné operace. Na konci fáze EX se v závislosti na typu instrukce může provést řízený výběr dat, která budou ve fázi WB zapsána do registru. Tyto data, signál řídící zápis do registru a adresa cílového registru určené pro stupeň WB se beze změny převedou do záchytného registru EX/WB a ve fázi WB se použijí k zápisu dat do odpovídajícího registru.

6.4 Technické prostředky pro eliminaci konfliktů

Závislosti mezi instrukcemi a konflikty, které tyto závislosti s sebou mohou přinášet, byly podrobně rozebrány v kapitole 3.2.2, proto je zde nebudu znovu všechny podrobně popisovat, ale zaměřím se jen na rozbor konfliktů, které se mohou vyskytnou u výše navržené linky pro zřetězené zpracování.

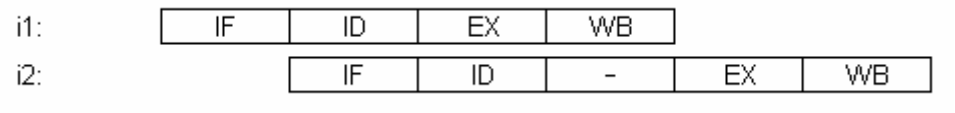
První skupinou závislostí, kterou se zde budu zabývat, tvoří závislosti datové. Tyto závislosti mohou být WAR (Write After Read, zápis po čtení), WAW (Write Aftrer Write, zápis po zápisu) a RAW (Read After Write, čtení po zápisu). Závislosti typu WAR se vyskytují při dokončování instrukcí mimo pořadí v programu a závislosti WAW se často objevují u instrukcí s pohyblivou řádovou čárkou, avšak u instrukcí s pevnou řádovou čárkou se nikdy nevyskytují. Protože navržená linka neumožňuje způsob zpracování instrukcí mimo pořadí ani nepracuje s instrukcemi s pohyblivou řádovou čárkou, nebudeme se závislostmi WAR a WAW dále zabývat. Zůstaly nám tedy pouze závislosti typu RAW, které vznikají následkem postupu zpracování dat v registrech a v paměti a které není možné zcela odstranit.

U navrženého zřetězeného zpracování existují dvě možnosti, jak může dojít ke vzniku konfliktu typu RAW. Jedná se buď o závislost načtení – použití, nebo závislost označovanou jako výpočet – použití. V našem případě není potřeba takto závislosti RAW dále rozlišovat, neboť jejich řešení bude stejné. Načtení i výpočet se totiž provádí ve stejné fázi. Návrh řešení konfliktu typu RAW tedy provedeme pomocí příkladu posloupnosti instrukcí, které přivedeme na vstup linky. Budeme uvažovat následující posloupnost instrukcí

i1: LD R1, R2

i2: ADD R3, R1

Instrukce i1 provádí načtení z paměti z adresy uložené v R2 a tyto data ukládá do registru R1, zatímco instrukce i2 používá registry R1 a R3 jako zdroje pro provedení operace sčítání a cílovým registrem pro uložení výsledku této operace je registr R3. Instrukce i2 ke svému provedení potřebuje data z registru R1 a nemůže být správně vykonána, dokud do registru R1 nezapíše data i1. Nejprve na obr. 6.2 ukážeme zpracování instrukcí i1 a i2 navrženou 4-stupňovou linkou, kde pozastavení linky je označeno znakem „–“.



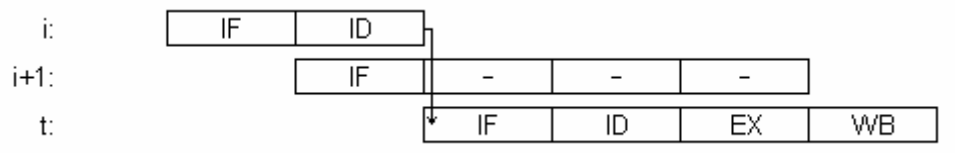
obr. 6.2 Diagram zpracování instrukcí i1 a i2 se zastavením linky

Jak je z obrázku patrné, vykonávání instrukce i2 musí být o 1 takt pozdrženo, než se stihne u instrukce i1 zapsat přečtená data do registrů. Při čtení dat z registru a zápisu dat do registru konflikt

nevznikne, neboť jsou uzpůsobeny k vykonání obou operací v jednom taktu a čtou se až aktuální data. Vzhledem k tomu, že přečtená data jsou k dispozici na výstupu datové paměti již na konci fáze EX, můžeme tyto data vzít a přenést je do fáze ID instrukce i2, kde se vloží do příslušného operandu, který se následně používá ve fázi EX. Tato technika se nazývá předávání dat (bypassing) a vyžaduje přidat do fáze ID multiplexor pro výběr dat prvního operandu a přidat sběrnice z výstupu ALU a datové paměti na vstupy multiplexorů obou operandů. Ve fázi ID se potom zjišťuje, zda identifikátory operandů nejsou shodné s identifikátorem cílového operandu uloženým v registru ID/EX. V případě použití techniky předávání dat se nám podaří vzniklé konflikty typu RAW vždy odstranit. Diagram na obr. 6.3 znázorňuje činnost linky při zpracování instrukcí i1 a i2 s použitím techniky předávání dat a šipka zde znázorňuje předávání dat.

obr. 6.3 Diagram zpracování instrukcí i1 a i2 s technikou předávání dat

Nyní se podíváme na chování navržené linky, když zpracovává instrukci skoku. Pokud se v programu objeví instrukce skoku, pak vzniká řídicí závislost mezi touto instrukcí a instrukcemi prováděnými po ní. Zpracování programu totiž probíhá zpracováváním následujících instrukcí až do té doby, než je k dispozici výsledek testu podmínky skoku, v případě podmíněném skoku, a cílová adresa, kam se má skočit. Navržené linka poskytuje tyto informace ve fázi dekódování skokové instrukce, tak že se vyhodnotí obsah podmínkového registru podle podmínky, která je součástí kódu instrukce, a zároveň vybere z kódu instrukce úplnou adresu cílové instrukce. Tím, že je celá adresa již součástí instrukčního kódu, nám sice odpadá činnost spojená s výpočtem nové adresy, ale nezabrání se tím načtení nesprávné instrukce v případě skoku. Chování instrukční linky při provedení skoku je znázorněno na obr. 6.4 a šipkou je znázorněn přenos adresy pro načtení instrukce t v případě, že se má skok provést.



obr. 6.4 Diagram zpracování instrukce skoku

Než se ve fázi ID instrukce i zjistí, že se má provést skok na adresu, kde je uložena instrukce t, načte se již další instrukce z adresy i+1. Načtení správné instrukce t proběhne se zpožděním 1 takt a vykonávání instrukce i+1 se zruší ve fázi ID. V případě, že by test podmínky byl negativní, dokončí se zpracování instrukce i+1 a další zpracování programu bude záležet na typu instrukce i+1. Podmínkové registry jsou nastavovány podle výsledků vytvořených při zpracování instrukce ve fázi

EX okamžitě, aby bylo možné testovat výsledek předchozí instrukce. Snížení pokut u provedených skoků lze dosáhnout implementací predikce skoků. Možnosti této techniky byly popsány v kapitole 3.3 a návrhu a implementaci do stávajícího procesoru bude věnována kapitola 7.

Poslední kategorie závislostí, strukturní závislosti, se vyskytují v případě, že více instrukcí potřebuje ve stejnou dobu využívat stejné prostředky. Konflikt, který se projevuje při přístupu do společné paměti pro data a instrukce z fází IF a EX, jsem odstranila již při návrhu samotného procesoru rozdělením paměti na datovou a instrukční podle Harvardské koncepce. Další potenciální konflikt se týká současného využití registrů, když se čtou data ve fázi ID a zároveň zapisují data ve fázi WB. Vzhledem k tomu, že navržený soubor registrů umožňuje v jednom taktu číst i zapisovat data, dokonce i když se tyto operace týkají stejného registru, je i tento konflikt odstraněn. Jediná situace, kdy na této lince může vzniknout konflikt prostředků, je výpadek v paměti cache, potom musí instrukce ve fázi ID čekat, než předchozí instrukce uvolní fázi EX.

7 Návrh predikce skoků

Toto je poslední kapitola, která se věnuje návrhu, a bude věnována návrhu opatření pro zvýšení výkonnosti stávajícího procesoru při vykonávání skokových instrukcí. Skokové instrukce se vyskytují v programu dost často, narušují sekvenční postup zpracování v instrukční lince a bez použití přidaných opatření výrazně snižují výkonnost linky. Proto bude tato kapitola věnována právě návrhu a implementaci takového prostředku, který pomůže eliminovat latence při provádění skokových instrukcí.

Pro navržený procesor se zřetězeným zpracováním instrukcí jsem se rozhodla pro zvýšení výkonnosti u skokových instrukcí použít metodu predikce skoků. Jednotlivé způsoby predikce jsem podrobně popsala v kapitole 3.3 a v tomto projektu jsem se rozhodla aplikovat dynamickou predikci použitím dvoubitového prediktoru skoků.

7.1 Dvoubitový prediktor skoků

Úkol tohoto mechanismu spočívá v odhadu cílové adresy skoku před tím, než je samotný skok proveden, aby umožnil výběr následující instrukce ze správné adresy. Využívá k tomu techniku dynamické predikce, kdy je predikce prováděna na základě předchozího chování daného skoku a historie dosavadního chování je reprezentována zvláštním záznamem pro každý skok tabulce historie skoků (BHT – Branch History Table).

Základ dvoubitového prediktoru tvoří tabulka historie skoků, kterou jsem implementovala jako plně asociativní paměť cache pro 4 položky. Klíč, podle kterého se v této paměti vyhledává, tvoří úplná adresa skoku. Další položky paměti už tvoří pouze záznam historie uloženého skoku, který zabírá v paměti 2 bity a informace o platnosti záznamu. Počet položek tabulky jsem volila, tak aby velikost tabulky byla přiměřeně menší ve srovnání s použitou velikostí cache v procesoru pro ukládání dat a instrukcí. Vzhledem k malému počtu položek a malým nárokům na prostor pro data jsem se rozhodla tuto paměť implementovat jako plně asociativní, kde je v případě potřeby k likvidaci vybrána nejdéle nepoužívaná položka podle algoritmu LRU.

Predikce probíhá tak, že se ve fázi IF vloží adresa, podle které se vybírá instrukce z instrukční paměti, také na vstup paměti cache prediktoru skoků. V tabulce BHT se provede porovnání vstupní adresy s uloženými záznamy. Pokud je záznam odpovídající této adrese v tabulce nalezen, jde o instrukci skoku a v následující fázi IF se další instrukce bude načítat podle predikce. Odpovídající adresu získáme v případě negativní predikce inkrementací adresy stávající instrukce, v případě pozitivní predikce výběrem cílové adresy skoku z kódu této instrukce. Pokud adresa v tabulce BHT nebyla nalezena, může se jednat o jinou než skokovou instrukci, nebo jde o instrukci skoku, která je buď prováděna poprvé, nebo byla dříve z paměti vymazána kvůli nedostatku prostoru. Zpracování

instrukce pokračuje dále ve fázi ID, kde zjistíme, zda se opravdu jednalo o instrukci skoku. Pokud se jednalo o instrukci podmíněného skoku zkontroluje se ještě test podmínky. Když je podmínka splněna a predikce byla pozitivní, pokračuje zpracovávání instrukce, která je momentálně ve fázi IF. Jestliže se ovšem liší výsledek podmínky od predikce, pak se skočilo špatným směrem, v další fázi IF musí být načtena správná instrukce a provádění původně načtené instrukce musí být zrušeno. Pokud se dekódováním zjistí, že se nejednalo o instrukci skoku, bude tato instrukce dále běžně zpracována. V případě, že byla ve fázi ID odhalena instrukce skoku, provede se v dalším taktu aktualizace záznamu u této instrukce podle skutečného chování. Jestliže adresa instrukce v paměti dosud nebyla zaznamenána, vloží se sem a historie skoku se inicializuje na hodnotu 11. Princip přechodů mezi jednotlivými stavy i význam jednotlivých stavů pro určení predikce byly popsány v kapitole 3.3 a vystihuje je tam diagram přechodů na obr 3.6.

8 Závěr

Tato kapitola bude věnována zhodnocení výsledků celé této práce. Budou zde nastíněny možnosti dalšího vývoje a zmíním se také o přínosech, které mi zpracování tohoto projektu přineslo.

Základním úkolem celého projektu byl návrh a implementace jednoduchého procesoru v jazyce VHDL, a tento byl splněn v rámci Ročníkového projektu. Návazně na tuto práci jsem se v Semestrální projektu věnovala studiu moderních prvků a principů, které by bylo možné využít ke zvýšení výkonnosti již navrženého procesoru. V této oblasti jsem se zaměřila na pochopení základních principů pamětí cache, metody pipelining a predikce skoků. V Diplomové práci jsem využila výsledky získané zpracováním předchozích projektů a pokračovala jsem návrhem již nastudovaných prvků a jejich postupnou implementací do výchozího procesoru. Vzniklo tak několik modifikací výchozího procesoru s odlišnými vlastnostmi. V poslední fázi této práce mělo být, na základě experimentálních výsledků získaných simulací, provedeno vzájemné porovnání vlastností vytvořených variant původního procesoru se zaměřením na rozdíly ve výkonnosti. Tuto část práce se mi ovšem nepodařilo splnit z důvodu neúspěšné realizace navržených procesorů.

Další vývoj tohoto projektu by se mohl ubírat několika směry. Je možné zaměřit se na oblast pamětí cache, zkusit implementovat paměti cache s různými parametry a porovnávat, jak jednotlivé typy ovlivňují výkonnost procesoru. Jiným směrem, kterým je možné se vydat, je úprava stávajícího skalárního procesoru na superskalární procesor s vydáváním více instrukcí v jednom taktu. Tato změna způsobu zpracování instrukcí s sebou přináší, kromě dalšího zvýšení výkonnosti, také spoustu problémů, které je třeba řešit při návrhu a implementaci, a proto si myslím, že nabízí velký prostor pro další vývoj tohoto projektu. Zajímavým pokračování tohoto projektu by mohla být také implementace navrženého procesoru do FPGA s tím, že by nejprve musely být kódy popisující tento procesor v jazyce VHDL upraveny tak, aby odpovídaly požadavkům na syntézu.

Přínosem této práce pro mě bylo to, že jsem se lépe zorientovala v oblasti architektur procesorů, a to zejména v části, která se věnuje prostředkům pro zvyšování výkonnosti procesorů.

Literatura

[1] Architektura počítačů, dokument dostupný na URL http://poli.cs.vsb.cz/edu/arp/down/pocitace.pdf (květen 2007).

[2] Dvořák, V., Drábek, V. Architektura procesorů, VUTIUM, Brno, 1999. 293 s. ISBN 80-214-1458-8.

[3] Podpora výuky hardware na bázi FPGA. Dokument dostupný na URL https://www.fit.vutbr.cz/study/courses/PCS/private/prednasky/06_navrh_procesoru/PavelFaltynekDiplomka.pdf (květen 2007).

[4] Pelikán, J. Hypertextový průvodce architekturou PC. Dokument dostupný z URL http://www.fi.muni.cz/usr/pelikan/ARCHIT/TEXTY/CACHE.HTML#LRU (květen 2007).

[5] Drábek, V. Výstavba počítačů, PC-DIR, Brno, 1995. 150 s. ISBN 80-214-0691-7.

[6] Hennessy, J.L., Patterson, D.A. Computer Architecture - A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, New York, 2003. 1000 s. ISBN 1-55860-596-7.

Seznam příloh

Příloha A

Instrukční soubor

označení operace	příznaky in		kód instrukce (oper. kód)																příznaky out	
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
NOP	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-	-
Prázdná operace																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ADD R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	0	0	0	√	√
Aritmetický součet registrů R_{DS1} a R_{S2}, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ADC R_{DS1}, R_{S2}	-	√	1	1	1	1	R_{DS1}				R_{S2}				0	0	0	1	√	√
Aritmetický součet registrů R_{DS1} a R_{S2} a vstupního přenosu C, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SUB R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	0	1	0	√	√
Aritmetický rozdíl registrů R_{DS1} a R_{S2}, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SBC R_{DS1}, R_{S2}	-	√	1	1	1	1	R_{DS1}				R_{S2}				0	0	1	1	√	√
Aritmetický rozdíl registrů R_{DS1} a R_{S2} a vstupního přenosu C, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
AND R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	1	0	0	√	-
Logický součin registrů R_{DS1} a R_{S2}, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.																				

označení operace	příznaky in		kód instrukce																příznaky out	
			oper. kód																	
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
OR R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	1	0	1	√	-

Logický součet registrů R_{DS1} a R_{S2}, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
XOR R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	1	1	0	√	-

Exkluzivní logický součet registrů R_{DS1} a R_{S2}, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
MOV R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				0	1	1	1	√	-

Přesun obsahu R_{S2} do registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
CMP R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				1	0	0	0	√	√

Porovnání obsahů registrů R_{DS1} a R_{S2}. Příznak zero nastaven, pokud jsou obsahy obou registrů shodné, jinak je nastaven příznak carry.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
LD R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				1	0	0	1	√	-

Přesun dat z datové paměti z adresy v R_{S2} do registru R_{DS1}. Příznak zero nastaven, pokud jsou data nulová.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ST R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				1	0	1	0	-	-

Přesun dat ze zdrojového registru R_{DS1} do datové paměti na adresu z R_{S2}.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
IN R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				1	0	1	1	√	-

Přesun dat z vstupního portu čísla v R_{S2} do registru R_{DS1}. Příznak zero nastaven, pokud jsou data nulová.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
OUT R_{DS1}, R_{S2}	-	-	1	1	1	1	R_{DS1}				R_{S2}				1	1	0	0	-	-

Přesun dat ze zdrojového registru R_{DS1} na výstupní port čísla z R_{S2}.

označení operace	příznaky in		kód instrukce														příznaky out	
			oper. kód															
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ADD R_{DS1}, *imm*	-	-	0	0	0	0	R_{DS1}				k	k	k	k	k	k	k	k	√	√

Aritmetický součet registruR_{DS1} a imm, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ADC R_{DS1}, *imm*	-	√	0	0	0	1	R_{DS1}				k	k	k	k	k	k	k	k	√	√

Aritmetický součet registru R_{DS1} a imm a vstupního přenosu C, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SUB R_{DS1}, *imm*	-	-	0	0	1	0	R_{DS1}				k	k	k	k	k	k	k	k	√	√

Aritmetický rozdíl registru R_{DS1} a imm, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SBC R_{DS1}, *imm*	-	√	0	0	1	1	R_{DS1}				k	k	k	k	k	k	k	k	√	√

Aritmetický rozdíl registru R_{DS1} a imm a vstupního přenosu C, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry je nastaven, pokud dojde k přenosu z nejvyššího řádu

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
AND R_{DS1}, *imm*	-	-	0	1	0	0	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Logický součin registru R_{DS1} a imm, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
OR R_{DS1}, *imm*	-	-	0	1	0	1	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Logický součet registru R_{DS1} a imm, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
XOR R_{DS1}, *imm*	-	-	0	1	1	0	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Exkluzivní logický součet registru R_{DS1} a imm, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

označení operace	příznaky in		kód instrukce														příznaky out	
			oper. kód															
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
MOV *R_{DS1}, imm*	-	-	0	1	1	1	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Přesun imm do registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
CMP *R_{DS1}, imm*	-	-	1	0	0	0	R_{DS1}				k	k	k	k	k	k	k	k	√	√

Porovnání obsahu registru R_{DS1} a imm. Příznak zero nastaven, pokud jsou shodné, jinak je nastaven příznak carry.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
LD *R_{DS1}, imm*	-	-	1	0	0	1	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Přesun dat z datové paměti z adresy v imm do registru R_{DS1}. Příznak zero nastaven, pokud jsou data nulová.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ST *R_{DS1}, imm*	-	-	1	0	1	0	R_{DS1}				k	k	k	k	k	k	k	k	-	-

Přesun dat ze zdrojového registru R_{DS1} do datové paměti na adresu z imm.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
IN *R_{DS1}, imm*	-	-	1	0	1	1	R_{DS1}				k	k	k	k	k	k	k	k	√	-

Přesun dat z vstupního portu čísla v imm do registru R_{DS1}. Příznak zero nastaven, pokud jsou data nulová.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
OUT *R_{DS1}, imm*	-	-	1	1	0	0	R_{DS1}				k	k	k	k	k	k	k	k	-	-

Přesun dat ze zdrojového registru R_{DS1} na výstupní port čísla z imm.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SL0 *R_{DS1}*	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	0	0	0	√	√

Posun registru R_{DS1} vlevo, na nejnižší bit registru R_{DS1} vložena nula, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.

	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SL1 *R_{DS1}*	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	0	0	1	√	√

Posun registru R_{DS1} vlevo, na nejnižší bit registru R_{DS1} vložena jednička, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.

označení operace	příznaky in		kód instrukce oper. kód														příznaky out	
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SLA R_{DS1}	-	√	1	1	0	1	R_{DS1}				0	0	0	0	0	0	1	0	√	√
Posun registru R_{DS1} vlevo, na nejnižší bit registru R_{DS1} vloženo carry, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SLX R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	0	1	1	√	√
Posun registru R_{DS1} vlevo, na nejnižší bit registru R_{DS1} vložen nejnižší bit, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ROL R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	1	0	0	√	√
Posun registru R_{DS1} vlevo, na nejnižší bit registru R_{DS1} vložen nejvyšší bit, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SR0 R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	1	0	1	√	√
Posun registru R_{DS1} vpravo, na nejvyšší bit registru R_{DS1} vložena nula, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SR1 R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	0	1	1	0	√	√
Posun registru R_{DS1} vpravo, na nejvyšší bit registru R_{DS1} vložena jednička, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SRA R_{DS1}	-	√	1	1	0	1	R_{DS1}				0	0	0	0	0	1	1	1	√	√
Posun registru R_{DS1} vpravo, na nejvyšší bit registru R_{DS1} vloženo carry, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
SRX R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	1	0	0	0	√	√
Posun registru R_{DS1} vpravo, na nejvyšší bit registru R_{DS1} vložen nejvyšší bit, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
ROR R_{DS1}	-	-	1	1	0	1	R_{DS1}				0	0	0	0	1	0	0	1	√	√
Posun registru R_{DS1} vpravo, na nejvyšší bit registru R_{DS1} vložen nejnižší bit, výsledek v registru R_{DS1}. Příznak zero nastaven, pokud je výsledek nulový. Příznak carry obsahuje vysouvaný bit.																				

označení operace	příznaky in		kód instrukce																příznaky out	
			oper. kód																	
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
JMP *adresa*	-	-	1	1	1	0	0	0	0	0	a	a	a	a	a	a	a	a	-	-
Nepodmíněný skok na adresu v programové paměti.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
JC *adresa*	-	√	1	1	1	0	0	0	0	1	a	a	a	a	a	a	a	a	-	-
Skok na adresu v programové paměti podmíněný platným příznakem carry.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
JZ *adresa*	√	-	1	1	1	0	0	0	1	0	a	a	a	a	a	a	a	a	-	-
Skok na adresu v programové paměti podmíněný platným příznakem zero.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
JNC *adresa*	-	√	1	1	1	0	0	0	1	1	a	a	a	a	a	a	a	a	-	-
Skok na adresu v programové paměti podmíněný neplatným příznakem carry.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
JNZ *adresa*	√	-	1	1	1	0	0	1	0	0	a	a	a	a	a	a	a	a	-	-
Skok na adresu v programové paměti podmíněný neplatným příznakem zero.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
CALL *adresa*	-	-	1	1	1	0	0	1	0	1	a	a	a	a	a	a	a	a	-	-
Nepodmíněný skok na adresu v programové paměti. Adresa této instrukce uložena do zásobníku návratových adres.																				
	zero	carry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	zero	carry
RET	-	-	1	1	1	0	0	1	1	0	-	-	-	-	-	-	-	-	-	-
Výběr adresy z vrcholu zásobníku návratových adres a skok na adresu po ní následující.																				