VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY
DEPARTMENT OF MICROELECTRONICS

PARALELISMUS NA ÚROVNI INSTRUKCÍ V MODERNÍCH PROCESORECH
INSTRUCTION LEVEL PARALLELISM IN MODERN PROCESSORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Pavel Sláma**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Marián Pristach, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Mikroelektronika**

Ústav mikroelektroniky

Student: Bc. Pavel Sláma ***ID:*** 173741
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Paralelismus na úrovni instrukcí v moderních procesorech

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s architekturou procesorů RISC a jejich instrukční sadou. Prozkoumejte možnosti paralelismu na úrovni instrukcí v moderních procesorech. Zaměřte se na techniky pro zpracování více instrukcí za takt a vykonávání instrukcí mimo programové pořadí. Vytvořte simulační model na RTL úrovni a implementujte zvolené techniky. Ověřte funkčnost modelu a změřte jeho výkon na vybrané sadě testů (Coremark, Dhrystone). Zhodnoťte přínos implementovaných technik na zvyšování výkonu procesoru a diskutujte možnosti pro jeho další nárůst.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Marián Pristach, Ph.D.
Konzultant: Bc. Marek Masařík, Codasip,s.r.o.

doc. Ing. Lukáš Fujcik, Ph.D.
předseda oborové rady

ABSTRAKT

Základní metodou pro dosažení paralelismu na úrovni instrukcí je metoda zřetězení linky používaná v procesorech již desítky let. Ideální zřetězená linka umožňuje zvýšit výkon a efektivitu procesoru za přidání jen malého množství zdrojů. Reálná zřetězená linka ale naráží na řadu limitací způsobených vzájemnými závislostmi mezi instrukcemi a dalšími faktory. Cílem této práce je diskutovat techniky používané pro zvyšování efektivity a výkonu procesoru se zřetězenou linkou, vybrané techniky implementovat na reálný model procesoru RISC a diskutovat jejich přínos.

KLÍČOVÁ SLOVA

Procesor, RISC, zřetězená linka, metoda vydávání více instrukcí za takt, Tomasulův algoritmus, Codasip Studio, CodAL, Codasip uRISC

ABSTRACT

Basic methodology that exploits instruction level parallelism is called pipelining and it is part of every processor for decades. The ideal pipeline increases performance and efficiency for a relatively small cost. But the real pipeline has number of limitations caused by dependencies and hazards between instructions. The aim of this thesis is to discuss techniques used to improve efficency and performance of pipelined processors, to implement selected techniques to a RISC processor model and discuss its benefits.

KEYWORDS

Processor, RISC, pipeline, multiple issue, Tomasulo algorithm, Codasip Studio, CodAL, Codasip uRISC

SLÁMA, Pavel. *Paralelismus na úrovni instrukcí v moderních procesorech*. Brno, 2020, 66 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce: Ing. Marián Pristach, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Paralelismus na úrovni instrukcí v moderních procesorech“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Marianu Pristachovi, Ph.D. za ochotu, trpělivost, odborné vedení a podnětné návrhy k práci. Dále bych rád poděkoval panu Bc. Marku Masaříkovi za cenné rady při návrhu a ladění modelů procesoru. Také bych chtěl poděkovat kolegům z firmy Codasip, přátelům, rodině a především své přítelkyni Andree za podporu při studiu a při vypracování této práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

V současné době jsou lidé obklopeni elektronikou více než kdy dříve. Ať už se jedná o televizory, osobní počítače, tablety nebo mobilní telefony, žádné z těchto a dalších zařízení, která jsou součástí našeho každodenního života, se již neobejde bez přítomnosti mikroprocesoru. Tyto vestavěné systémy cílí na využití aplikačně specifických procesorů pro dosažení vysokého výpočetního výkonu, nízké spotřeby a především nízké ceny. Trendy v oblasti vestavěných systémů jsou v posledních letech například zařízení pro tzv. internet věcí či nositelná elektronika. Neustále narůstající požadavky spotřebitelů kladou větší nároky na parametry použitých komponent, jímž vévodí procesory.S narůstajícími nároky na výkon procesorů je potřeba přinášet nová řešení, která pomáhají zvyšovat efektivitu procesoru.

Společnost Codasip, s.r.o. se zabývá návrhem mikroprocesorů pro vestavěné systémy za využití automatizovaného procesu návrhu ve vlastní unikátní sadě nástrojů Codasip Studio a jazyka pro popis architektur CodAL. Motivací této práce je prozkoumat řešení používaná pro navýšení výkonu v moderních procesorech a vyhodnotit jejich benefity. Zároveň také prozkoumat možnosti jazyka CodAL a zjistit, zda-li obsahuje dostatečné množství zdrojů i pro implementaci náročnějších architektur a odhalit případné limitace, které by bránily dalšímu vývoji produktového portfolia společnosti.

1 Architektura mikroprocesoru

Mikroprocesor je označení pro centrální procesorovou jednotku, jež je integrována jako součástka do pouzdra integrovaného obvodu. Centrální procesorová jednotka, často označována jako CPU (z angl. *Central processing unit*), je programovatelný elektrický obvod, který slouží pro vykonávání algoritmů za pomocí definovaných elementárních operací (instrukcí). Každý mikroprocesor je specifický svojí realizací, kterou lze klasifikovat použitou instrukční sadou, architekturou či zaměřením.

1.1 Instrukční sada

Instrukční sada (ISA) je abstraktní model počítače, který slouží jako rozhraní mezi hardwarem a softwarem a definuje jakým způsobem je program zpracován, tedy jaké operace je potřeba provést při kompilaci programu a jaké operace musí provést samotný procesor pro vykonání programu. ISA deklaruje prostředky a mechanismy pro programování, mezi které patří například seznam dostupných instrukcí a datových typů nebo architektonické zdroje jako například seznam registrů, pravidla pro řešení výjimek a přerušení, způsoby přistupování do paměti a další.

CISC

Komplexní instrukční sada (anglicky *Complex Instruction Set Computing*) je tvořena spoustou (řádově stovkami) instrukcí, které mají obvykle velmi specifickou funkci. Instrukce mají různou délku, vyznačují se efektivním, avšak mnohdy pomalejším přístupem do paměti a mohou pracovat s operandy v registrovém poli procesoru i v paměti. Procesor CISC tak obsahuje velké množství způsobů adresování paměti. S narůstajícím počtem instrukcí jsou však kladeny vysoké nároky na překladač, pro který je obtížné využít všechny dostupné instrukce. Ve výsledku CISC procesory nabízí široké spektrum zdrojů pro zpracovávání libovolných algoritmů, což je však kompenzováno celkovou složitostí hardwaru. [1]

RISC

Redukovaná instrukční sada (anglicky *Reduced Instruction Set Computing*) obsahuje menší počet obecněji zaměřených instrukcí. V programu je tedy jedna složitější instrukce (v případě CISC) rozdělena na více jednodušších instrukcí, které jsou postupně zpracovány pro dosažení stejného výsledku. Instrukce mají obvykle pevně danou šířku a několik definovaných formátů pro snížení nároků na jejich dekódování. Datové operace jsou prováděny pouze nad operandy v registrovém poli procesoru a pro přístup do paměti jsou definovány samostatné instrukce. Procesor RISC

také obsahuje relativně malé množství adresovacích módů. Koncept RISC procesoru tímto cílí na dosažení stejného nebo vyššího výpočetního výkonu při menší ploše a spotřebě, tedy při nižších výrobních a provozních nákladech. [1]

1.2 Architektura

Harvardská architektura

Harvardská architektura fyzicky odděluje paměť pro program a data. Hlavní výhoda tkví v paralelním přístupu do obou pamětí, což navyšuje rychlost načítání dat z obou paměťových prostorů. Dalším přínosem oddělených pamětí je fakt, že obě paměti mohou mít naprosto odlišné parametry, například šířku slova, způsob adresování nebo časování a technologii. Některé zabezpečené systémy tohoto využívají a pro paměť programu používají paměti typu ROM, které lze pouze číst, ale nelze modifikovat jejich obsah.

Von Neumannova architektura

Kontrastem k harvardské architektuře je model popsaný matematikem a fyzikem Johnem von Neumannem. Tento model popisuje počítač se sjednoceným paměťovým prostorem pro program i data, který zároveň používá společnou systémovou sběrnici pro všechny aktivní prvky - CPU, paměť, vstupy a výstupy a řídící jednotku. Společný adresní prostor přináší výhodu pro programátora. Nevýhodou však je sekvenční čtení instrukcí i dat po společné sběrnici, která má omezenou datovou propustnost a při neustálém navyšování výkonu se stává prvkem limitujícím maximální výkon této architektury.

Modifikovaná harvardská architektura

V současnosti je nejrozšířenější architektura v literatuře označována jako modifikovaná harvardská architektura. Ve skutečnosti jde o architekturu, jenž kombinuje obě předchozí varianty a profituje z jejich výhod. Základem jsou dvě oddělená rozhraní, která dovolují načítání instrukce i dat zároveň. Rozdílem oproti harvardské architektuře je rozvolnění pravidla na oddělený adresní prostor. V praxi je toho docíleno společnou pamětí pro program i data, přičemž mikroprocesor k ní přistupuje skrze separátní instrukční a datové rozhraní, každé z nich doplněné o rychlou vyrovnávací paměť, tzv. paměť *cache*. [2]

1.3 Zaměření procesoru

Univerzální procesory

Univerzální procesory jsou navrženy tak, aby mohly vykonávat široké spektrum operací. Běžně se vyskytují například v osobních počítačích, kde je žádoucí nabídnout univerzální využití za cenu nižšího výkonu oproti specializovaným procesorům.

Aplikačně specifické procesory

Zaměření aplikačně specifických procesorů je úzce omezeno na konkrétní funkci. Vlivem této optimalizace dokáží aplikačně specifické procesory dosáhnout mnohem vyšší efektivity při použití pro danou aplikaci. Nevýhodou užšího zaměření procesoru bývá složitější proces návrhu a výroby, což vede k vyšším finančním nákladům. [3], [4] Mezi aplikačně specifické procesory se řadí například:

- Digitální signálové procesory, které jsou primárně určeny ke zpracování číslicových signálů. Vynikají vysokou propustností operací pro aritmetické operace nad celými čísly a čísly s pevnou i plovoucí řádovou čárkou. Kladou však velké nároky na rychlost použitých pamětí.
- Aplikačně specifické instrukční procesory, jenž se vyznačují velmi úzce zaměřenou instrukční sadou. Instrukční sada i architektura procesoru jsou optimalizovány pro konkrétní specifickou aplikaci, kde tyto procesory dokáží dosáhnout velmi vysoké efektivity.
- Aplikačně specifické integrované obvody, které nejsou programovatelné a realizují konkrétní algoritmus na úrovni hardwaru.

2 Paralelismus na úrovni instrukcí

Pro zvyšování výpočetního výkonu procesoru je výhodné provádět více instrukcí zároveň. Základním způsobem dosažení vykonání více instrukcí za takt je metoda zřetězení linky procesoru. Tento princip však není ideální a má svoje limitace. K vylepšování zřetězené linky se využívá nespočet implementačních technik, které se snaží docílit dalšího navyšování výkonu a řeší nebo i částečně obcházejí některé problémy způsobené zřetězením. V tabulce 2.1 jsou shrnuty některé implementační techniky v praxi používané pro navyšování výkonu procesoru se zřetězenou linkou, z nichž některé jsou diskutovány v této kapitole. Tato kapitola včetně podkapitol čerpá množství informací z knih [5], [6] a [4].

Tab. 2.1: Přehled některých implementačních technik pro navyšování výkonu procesoru

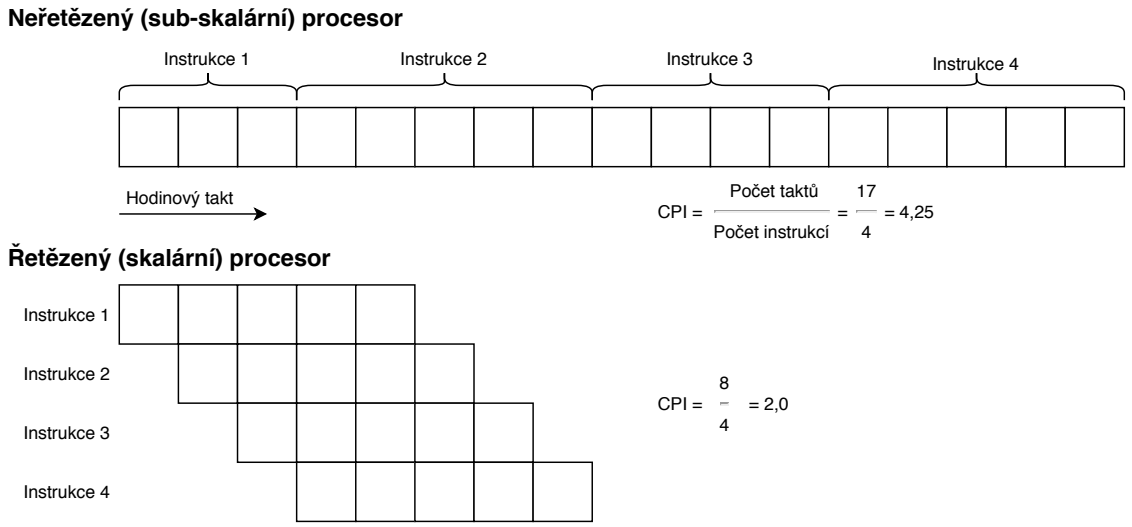
Implementační technika	Cíle
Zřetězené zpracování instrukcí	Vyšší IPC - nižší CPI
Forwarding	Redukce datových hazardů
Predikce větvení	Redukce řídících hazardů
Dynamické plánování instrukcí	Redukce datových závislostí
Spekulace na hardwarové úrovni	Redukce datových i řídících závislostí
Pokročilé dynamické plánování s přejmenováním registrů	Redukce pravých i nepravých hazardů
Vydávání ke zpracování více instrukcí za takt	Dosažení ideálního IPC, CPI
Vláknový paralelismus	Maximalizace využití výpočetních jednotek

2.1 Zřetězené zpracování instrukcí

Zřetězené zpracování (angl. *pipelining*) je technika běžně používána již od 80. let minulého století. Použitím zřetězené linky lze dosáhnout značného navýšení výkonu procesoru za přidání jen malého množství hardwaru. Procesor se synchronní zřetězenou linkou je také schopný pracovat na vyšším kmitočtu, což napomáhá k dalšímu navýšení výpočetního výkonu.

Základní myšlenkou zřetězení je rozdělení zpracování jedné instrukce na více dílčích kroků, které je možné provést samostatně. Každý krok se nazývá stupněm nebo také fází linky (angl. *pipeline stage*). K úplnému vykonání instrukce je třeba její průchod všemi fázemi linky. Paralelismu je dosaženo tak, že linka je naplněna instrukcemi v různých fázích rozpracování. Cílem zřetězení je dosáhnout větší propustnosti procesoru, tedy dosáhnout většího množství zpracovaných instrukcí za jeden takt, při zachování stejné instrukční latence. Nutné je však rozdělit všechny instrukce na stejný počet operací, jelikož počet stupňů linky je neměnný. První zřejmou nevýhodou zřetězení je fakt, že instrukce u nezřetězeného (sub-skalárního)

procesoru mohou mít různou latenci. U zřetězeného (skalárního) procesoru musí všechny instrukce projít stejnou linkou, je tedy nutné aby se počet kroků potřebný k jejich zpracování vyrovnal s délkou linky. Na obrázku 2.1 je tato situace znázorněna na příkladu čtyř instrukcí s různou latencí. Tyto čtyři instrukce poté putují pěti stupňovou linkou.



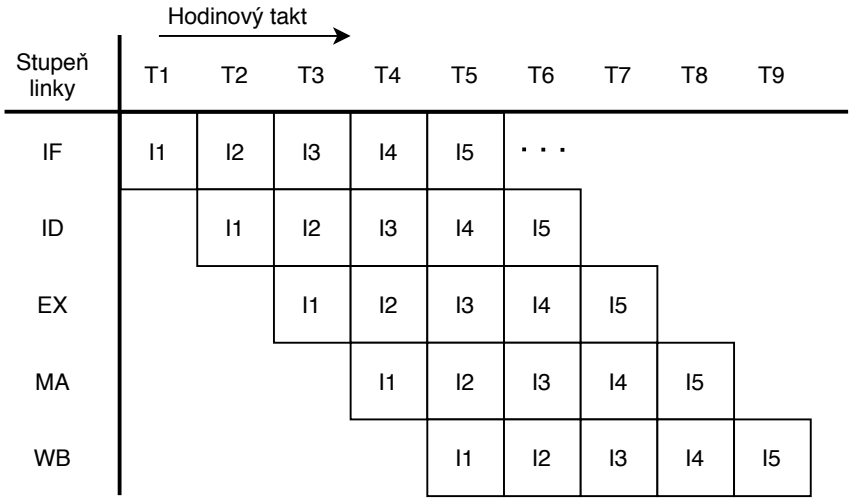
Obr. 2.1: Příklad přínosu zřetězeného zpracování instrukcí

Průměrná doba vykonání jedné instrukce nebo také CPI (z angl. *Cycles per instruction*) je pak v tomto konkretním případě u sub-skalárního procesoru více než dvojnásobná oproti skalárnímu. Přidáváním dalších instrukcí do výpočtu by docházelo k dalšímu snižování CPI zřetězené linky až k ideální hodnotě, kdy je CPI rovno jedné.

Běžný procesor typu RISC se obvykle skládá z pětistupňové zřetězené linky, která často vychází z výukového procesoru DLX. Linka procesoru DLX (obr. 2.2), který navrhli profesoři John L. Hennessy a David A. Patterson v devadesátých letech minulého století, se skládá z následujících stupňů:

- **Instruction fetch (IF)** - *Načtení instrukce.* V tomto stupni se na základě hodnoty uložené v programovém čítači přečte instrukce z paměti programu a na základě velikosti instrukce se poté inkrementuje programový čítač.
- **Instruction decode (ID)** - *Dekódování instrukce.* V tomto stupni se dekóduje instrukce. Pokud je to možné, počítá se zde cílová adresa skoku nebo větve a případně se provádí predikce. Zároveň se také načítají hodnoty operandů z registrového pole, což je možno provést paralelně k dekódování, protože RISC instrukce disponují pevným formátem.

- **Execution (EX)** - *Vykonání instrukce*. Stupeň vykonání instrukce je vybaven aritmeticko-logickou jednotkou (ALU), která dle typu instrukce provádí výpočet adresy skoku, vyhodnocení podmínky pro větev nebo aritmetickou operaci nad danými operandy.
- **Memory access (MA)** - *Fáze přístupu do paměti*. V tomto stupni se vykonávají LOAD a STORE instrukce pro načtení, respektive zápis dat.
- **Write-back (WB)** - *Zápis výsledků*. Poslední stupeň linky slouží k zápisu výsledků. Pokud instrukce provádí aritmetický výpočet a nebo načítá data z paměti, výsledek této instrukce se v tomto stupni zapisuje do registrového pole.



Obr. 2.2: Zpracování instrukcí v jednoduché zřetězené lince procesoru DLX

2.1.1 Hazardy

Hazard je situace, kdy následující instrukce ve zřetězené lince nemůže být v dalším taktu vykonána, čímž dochází ke snižování výkonu zřetězené linky. Vykonání takové instrukce by potenciálně znamenalo provedení nesprávného výpočtu. Hazardy vznikají v lince z různých důvodů, dle nichž je dělíme do tří skupin na datové, strukturální a řídící. Eliminovat hazardy je možné například pozastavením části zřetězené linky (angl. *stall*) a vkládáním "prázdných"operací. Tímto způsobem ale dochází ke snižování propustnosti linky. Některým závislostem lze také jednoduše předejít už při překladu programu, kdy přeskupením instrukcí může dojít k eliminaci hazardu, ovšem přesun instrukce nesmí porušit sémantiku programu.

Datové hazardy

Datový hazard vzniká v okamžiku, kdy ve zřetězené lince dochází ke změně pořadí čtení nebo zápisu hodnoty některého z operandů vůči pořadí, ve kterém by z pohledu instrukce nastaly při nezřetězeném zpracování. Pokud existují v lince dvě instrukce operující nad daty ze stejného zdroje, čili ze stejného registru nebo adresy v paměti, mohou nastat tři situace způsobující datový hazard.

- **Čtení následující zápis (RAW)** - vzniká v okamžiku, kdy aktuální instrukce čte hodnotu ze zdroje, který má být modifikován některou z předchozích instrukcí v lince a výsledek ještě není zapsán. V takovém případě by aktuální instrukce použila neplatnou hodnotu a musí tedy čekat na zápis správného výsledku.
- **Zápis následující čtení (WAR)** - nastává, když následující instrukce zapisuje data do zdroje, ze kterého aktuální instrukce čte. Dojde-li k zápisu dat následující instrukce dříve, než ke čtení aktuální instrukce, data v tomto zdroji budou neplatná.
- **Zápis následující zápis (WAW)** - je způsoben, pokud dvě různé instrukce zapisují data do stejného zdroje. Pokud se vykoná druhý zápis dříve než první, výsledná hodnota tohoto zdroje bude po dokončení obou instrukcí neplatná.

První hazard (RAW) se vyskytuje ve zřetězené lince nejčastěji. Zbylé dva hazardy (WAR a WAW) ovšem nemůžou nastat v jednoduchých linkách, které vykonávají instrukce v programovém pořadí. K jejich výskytu dochází až u složitějších procesorů, kde se některé instrukce mohou předbíhat a vykonávat mimo programové pořadí.

Strukturální hazardy

Ke strukturálnímu hazardu v lince dojde, pokud se dvě instrukce ve stejný okamžik snaží přistoupit ke stejné jednotce (např. ALU), která však nepodporuje vícenásobný přístup. Tento hazard je možné eliminovat například pozastavením linky nebo přidáním dalších jednotek.

Řídící hazardy

Pokud se v lince nachází instrukce měnící běh programu, tedy instrukce pro podmíněný nebo nepodmíněný skok, a v prvním stupni linky není možno rozhodnout zda bude skok proveden a na jakou adresu, dochází k řídícímu hazardu. Důsledkem je, že v případě provedení skoku je třeba vyjmout z linky všechny rozpracované instrukce a začít linku plnit od znovu z nové adresy. Pokutu za provedení skokové instrukce je možné eliminovat, pokud se procesor pokusí předpovědět výsledek skokové instrukce. Tato technika se souhrnně označuje jako predikce skoků.

2.2 Forwarding

Pro řešení datových hazardů tak, aby nedocházelo ke snižování výkonu zřetězené linky jejím pozastavením, se používá jednoduchá technika nazývaná forwarding (případně také *bypassing*). Hlavní myšlenkou této metody je, že není potřeba pozastavovat linku ve fázi načítání operandů z registrového pole, protože v tuto chvíli instrukce ještě nepotřebuje znát hodnotu operandu. Místo pozastavení se tato instrukce nechá putovat linkou až do místa provedení operace, kde je již nutné znát hodnotu operandu. V tuto chvíli je ale předchozí instrukce, která má zapsat výsledek operandu této aktuální instrukce, již v dalších stupních linky a je-li výsledná hodnota operandu už známa, tak se tato hodnota posune přímo aktuální instrukci, tedy ještě před zápisem do registrového pole. Tímto dochází k obcházení ("bypassu") registrového pole - odtud alternativní název *bypassing*.

Pro eliminaci možných prostojů linky se zavadí zkratky ze všech stupňů produkujících výsledky, které ještě nejsou zapsány do registrového pole zpět do jednotek, kde je nezbytně nutné znát hodnoty operandů. Tyto cesty se aktivují logikou, která kontroluje zda jsou operandy k dispozici z dřívějších stupňů linky nebo zda jsou operandy k dispozici na některé ze zkratek. Pokud nejsou operandy k dispozici ani na žádné ze zkratek, pak je opět nutné pozastavit linku a počkat, než bude hodnota vypočítána.

2.3 Predikce skoků

Řídící hazardy obecně způsobují mnohem větší výkonové ztráty ve zřetězené lince než datové hazardy a pravidlem je, že čím je zřetězená linka delší, tím větší pokutu řídící hazardy způsobí. Pro moderní výkonné procesory s dlouhou zřetězenou linkou je proto důležité řešit řídící hazardy co možná nejefektivněji.

Skoky v RISC procesorech se dají obecně rozdělit na podmíněné či nepodmíněné skoky. Nepodmíněné skoky se provedou vždy, zatímco podmíněné pouze při splnění podmínky. U nepodmíněných skoků je snaha předpovídat cíl skoku dříve než je vypočítaná skutečná adresa, zatímco u podmíněných skoků se předpovídá, zda se skok provede či nikoliv. Predikce skoků se dle způsobu realizace rozděluje na statickou a dynamickou.

2.3.1 Statická predikce

Metody statické predikce jsou obecně velmi jednoduché a cílí na minimální nárůst složitosti hardwaru. Řešení jsou různá a na úspěšnost predikce má velký vliv optimálnost návrhu vykonávaného programu.

Jako možný způsob statické predikce lze například označit přístup, kdy se předpokládá, že se žádný podmíněný skok nevykoná a linka se plní dál. K penaltě za opětovné naplnění linky pak dochází pouze v případech, kdy se skok vykoná. Dalším principem je například predikce smyček, kdy se předpokládá, že skoky mířící zpět v programu (do další iterace smyčky) se vykonají vždy a skoky mířící opačným směrem se nevykonají nikdy. Taková predikce je výhodná v algoritmech s velkým množstvím smyček, kdy se pouze při poslední iteraci smyčky skok nevykoná a predikce je nesprávná.

2.3.2 Dynamická predikce

Dynamické prediktory používají sofistikovanější metody predikce a mnohdy kombinují více různých přístupů pro dosažení vyšší efektivity, ovšem za cenu nárůstu logiky. Hlavní myšlenkou je sledování historie vykonávání programu, odhadování skoků, následné vyhodnocení a korekce předpovědi. Předpokladem je, že pokud se skrze stejnou větev programu prochází opakovaně, tak každý další běh programu je vykonáván s lepší úspěšností predikce.

Problematika dynamické predikce je velmi rozsáhlá, základním a nejjednodušším principem je prediktor zvaný saturační čítač, který slouží pro předpověď, zda se skok provede. Tento prediktor se často kombinuje s různými zásobníky, které uchovávají cílovou adresu skoku, např. *branch target buffer* a *return address stack*.

Branch target buffer

Branch target buffer nebo-li zásobník cílových adres uchovává poslední adresy vykonaných skoků. Pokud prediktor pro konkrétní adresu předpoví, že se skok vykoná, tento zásobník poskytne cílovou adresu. Pokud je predikce správná a zásobník obsahuje správnou cílovou adresu, linka se nepřetržitě plní validními instrukcemi a nedochází k žádným pokutám. Tento zásobník se obvykle používá pro určení cíle podmíněných skoků a nepodmíněného skoků (kromě návratu z podprogramu).

Return address stack

Return address stack, zásobník návratových adres, pracuje se skoky, které slouží pro volání podprogramu. Taková skoková instrukce, která je použita k volání podprogramu, je obvykle doplněna druhou instrukcí na konci podprogramu, která se vrací do hlavní smyčky na instrukci, která následuje instrukci volání podprogramu. Při vykonání skoku pro volání podprogramu se uloží návratová hodnota do zásobníku a při návratu je k dispozici přesná návratová adresa, čímž nedochází k žádné pokutě způsobené řídícím hazardem.

N-bitový saturační čítač

N-bitový saturační čítač (angl. *saturating counter*) je také v anglické literatuře označován jako *branch history table* nebo *branch-prediction buffer*.

Základní prediktor je vybaven malou pamětí, kde každé slovo má šířku pouze jednoho bitu. Dle počtu položek se poté použije malý kousek adresy, tzv. index, pro adresování paměti prediktoru a v té je uložena informace, zda byl či nebyl podmíněný skok už někdy vykonán. Výhoda tkví v jednoduché implementaci. Ovšem nevýhodou může být, že více různých adres má stejný index, tudíž je predikce nepřesná a ovlivněná i jinými skoky se stejným indexem. Také skok na stejné adrese, který se vykoná téměř vždy, pokaždé změní hodnotu predikce a tudíž se pravděpodobně splete ve dvou případech při každé v posloupnosti vykonávání, čímž se opět snižuje její přesnost.

Pro vylepšení vlastností tohoto prediktoru se používá dvoubitový čítač. Základní princip je stejný, pouze je šířka slova pro každý index rozšířena na dva bity, tudíž má predikce čtyři stavy namísto dvou u jednobitového čítače. Výhodou dvou bitového schématu je, že se musí výsledek odchýlit dvakrát od skutečného vykonání než se změní predikce. Úspěšnost predikce dvou bitového čítače se 4096 položkami se dle testovací sady SPEC89 pohybuje mezi 82 až 99 procenty, v závislosti na konkrétním algoritmu. Dalšího potenciálního zlepšení lze dosáhnout navýšením počtu položek, přičemž průměrná úspěšnost predikce na celé sadě SPEC89 při zvětšování počtu položek saturuje na 93,5 procentech. Pro svou relativní jednoduchost a poměrně dobrou efektivitu je dvou bitový čítač často používaný a to i u složitějších procesorů v kombinaci s některým dalším prediktorem.

2.4 Dynamické plánování instrukcí

Základní myšlenkou zřetězené linky je vykonávání instrukcí v programovém pořadí (*in-order execution*). Tento přístup přináší značné limitace ve chvíli, je-li linka pozastavena kvůli nemožnosti vykonat okamžitě aktuální instrukci. Vykonání každé následující instrukce musí čekat dokud se linka znovu neuvolní. Pokud jsou tedy po sobě jdoucí instrukce na sobě závislé, dochází ke vzniku hazardu a linka musí být často pozastavena. Zároveň pokud procesor obsahuje více výpočetních jednotek, tyto ostatní jednotky také čekají na uvolnění linky a nepracují. Tato nevýhoda se dále značně prohlubuje, je-li linka pozastavena instrukcí s velmi dlouhou latencí - například instrukcemi pro celočíselné dělení nebo dělení s plovoucí desetinnou čárkou, jejichž latence může dosahovat až desítky hodinových cyklů. Představíme-li si například instrukci pro dělení s dlouhou latencí následovanou instrukcí závislé na výsledku dělení a poté řadu nezávislých instrukcí, z hlediska maximalizace efektivity

využití výpočetních jednotek se jeví jako výhodné "předběhnout"při vykonávání instrukci dělení a vypočítat výsledky dalších nezávislých instrukcí. Takovému přístupu se říká vykonávání instrukcí mimo programové pořadí (*out-of-order execution*).

2.4.1 Vykonávání instrukcí mimo programové pořadí

Vykonávání instrukcí mimo programové pořadí přináší již diskutované benefity pro zvyšování výkonu procesoru. Zároveň také ale přináší nové konflikty a problémy do zřetězené linky, se kterými je potřeba se při návrhu linky vypořádat, jinak by procesor mohl produkovat nesprávné výsledky. Vykonávání instrukcí a především zápis výsledků mimo programové pořadí vnáší do linky hazardy WAR a WAW, které již byly diskutovány v sekci 2.1.1. Další komplikace nastávají při zachycení a zpracování výjimek v průběhu vykonávání programu. I přes vykonávání instrukcí mimo programové pořadí je totiž třeba zpracovávat výjimky tak, jako by byly instrukce vykonávány ve správném pořadí. Tato situace se dá vyřešit pozastavením výjimky před zápisem instrukce a čekáním, dokud by tato instrukce nebyla na řadě v programovém pořadí.

Avšak i pokud procesor obsluhuje výjimky tímto způsobem, stále může docházet k tzv. nepřesným výjimkám (z angl. *imprecise exception*). Nepřesná je taková výjimka, která nastane při vykonávání instrukcí mimo programové pořadí, přičemž při vykonávání v programovém pořadí by nenastala. Nepřesné výjimky vznikají v lince v situacích, kdy jsou již zapsány výsledky některých následujících instrukcí, přičemž aktuální instrukce způsobí výjimku, nebo naopak aktuální instrukce působí výjimku ale v lince se ještě nacházejí některé předchozí instrukce, jejichž výsledky ještě nebyly vypočítány a zapsány. Problém nepřesných výjimek tkví především v návratu z obsluhy výjimky a pokračování ve vykonávání programu. Například dochází-li při návratu z obsluhy výjimky ke skoku zpět do hlavního programu na adresu instrukce, jež způsobila výjimku, dojde k tomu, že předchozí instrukce, jejichž výsledky se ještě nezapsaly, se už nevykonají a následující instrukce, jejichž výsledky se už zapsaly, se mohou vykonat znovu. Tento problém způsobuje odchýlení od sémantiky programu a opět vede k nesprávným výsledkům.

2.4.2 Implementace dynamického plánování

Pro dynamické plánování instrukcí se v praxi používá více postupů. Jednou ze známějších metod je algoritmus zvaný *scoreboarding*, který byl poprvé použit v počítači CDC 6600 v roce 1964. Cílem metody scoreboarding je v případě výpadku linky z důvodu datového hazardu zvýšit výkon architektury vykonáním některé následující instrukce, existuje-li ve frontě taková která má všechny operandy připraveny. Algoritmus tedy dovoluje vykonávání instrukcí mimo programové pořadí, avšak je

poměrně jednoduchý na implementaci a nevyžaduje velké množství zdrojů. I proto se v současnosti stále používá, avšak spíše v jednodušších aplikacích, které cílí na malou plochu čipu a malou spotřebu. Pro výkonnější aplikace se používá pokročilejší algoritmus plánování instrukcí, který byl poprvé použit již v roce 1966 společností IBM v počítači s označením System/360 Model 91. Pro jeden z nejvýkonnějších počítačů své doby, jež k výpočtům používal i Národní úřad pro letectví a kosmonautiku (NASA), byla stěžejní jednotka matematického koprocesoru sloužící pro výpočty nad čísly s plovoucí řádovou čárkou. Matematický koprocesor (zkráceně FPU z angl. *floating-point unit*) dokázal zpracovat až 16,6 milionů instrukcí za sekundu. Aby FPU jednotka dosahovala takto vysokého výkonu, používala zřetězenou linku a algoritmus pro dynamické plánování instrukcí pro jejich vykonávání mimo programové pořadí. Algoritmus byl po svém vynálezci Robertu Tomasulovi pojmenován jako Tomasulův algoritmus a je i v současnosti stěžejním prvkem výkonných procesorů.

2.4.3 Scoreboarding

Scoreboarding je metoda pro dynamické plánování instrukcí, která umožňuje vykonávání instrukcí mimo programové pořadí, pokud neexistuje žádný hazard, který by bránil správnému vykonání instrukcí.

Základem algoritmu je tabulka skóre (angl. *Scoreboard*), která uchovává informace o datových závislostech pro každou instrukci a také informaci, která funkční jednotka bude instrukci vykonávat. Instrukce jsou ve zřetězené lince načteny a dekódovány v programovém pořadí. Poté následují čtyři kroky potřebné pro vykonání instrukce - vydání (*Issue*), čtení operandů (*Read operands*), vykonání instrukce (*Execution*) a zapsání výsledků (*Write results*). V prvním kroku se kontrolují datové závislosti. Zjišťují se zdrojové a cílové registry dané instrukce. Pokud dochází k výstupní závislosti, tedy některá z předchozích instrukcí zapisuje do cílového registru (WAW hazard), vykonání instrukce se odkládá dokud závislost nezanikne. Nachází-li se v tabulce skóre instrukce, která nemá žádné datové závislosti a zároveň je dostupná funkční jednotka pro její vykonání, vydává se tato instrukce ke zpracování. Poté, co byla instrukce vydána a byla zarezervovaná funkční jednotka, přechází se ke čtení zdrojových registrů. V tomto kroku se kontrolují vstupní závislosti z důvodů předcházení RAW hazardů. Má-li některá předchozí instrukce zapsat do jednoho ze zdrojových registrů aktuální instrukce, tento operand se označí jako nedostupný, dokud se nezapíše výsledek, a vykonání této instrukce se pozastavuje. Po nachystání všech operandů dochází k vykonání instrukce. Poté, co je znám výsledek operace, aktualizují se informace v tabulce skóre a přechází se k poslednímu kroku. Tím je zápis výsledků do registrového pole. Tento proces je však pozastaven do doby, než

všechny dřívější instrukce prochází druhým krokem - čtením operandů. Poté, co se v tabulce skóre nenachází žádná dřívější instrukce, která má zdrojový registr stejný jako je cílový registr aktuální instrukce, již nevzniká WAR hazard a může se zapsat výsledek.

Scoreboarding je v principu velmi jednoduchý algoritmus, který ale nedosahuje maximální efektivity. Největší slabinou je řešení konfliktů, kdy se vždy pozastavuje vykonávání instrukce, která nějaký hazard způsobuje, a tím vznikají prostoje v lince. Mezi výhody patří již zmíněná jednoduchost, kdy je pro realizaci potřeba méně prostředků. *Scoreboarding* se tedy hodí pro procesory, jenž cílí na nízkou spotřebu a malé výrobní náklady spíše než na vysoký výpočetní výkon.

2.4.4 Thomasulův algoritmus

Na rozdíl od metody scoreboarding cílí Tomasulův algoritmus na maximální efektivitu využití funkčních jednotek. Za pomocí několika metod, kterými jsou především metoda přejmenování registrů, rezervační stanice a společná datová sběrnice, se snaží eliminovat maximum hazardů a dosáhnout minimálních prostojů u každé funkční jednotky paralelním zpracováním instrukcí. Základním principem algoritmu je načítání a dekódování instrukcí v programovém pořadí. Také vydávání instrukcí se provádí v programovém pořadí, čímž je umožněno provádět kontrolu různých závislostí mezi instrukcemi, což částečně pomáhá eliminovat generování nepřesných výjimek.

Metoda přejmenování registrů slouží k vytvoření abstraktní vrstvy mezi logickými (architekturálními) registry a fyzickými registry. Strojové instrukce operují s množinou registrů definovaných v instrukční sadě. Většina instrukčních sad definuje ale pouze malé množství registrů, např. sada x86-64 definuje 16 registrů pro celočíselné operace a procesory s redukovanou instrukční sadou obsahují obvykle 32 takových registrů. Z důvodu relativně malého množství dostupných registrů dochází k častým datovým závislostem mezi instrukcemi, které nejsou na sobě přímo závislé, ale používají stejné registry a tudíž dochází ke vzniku některého z hazardů. Pomocí přejmenování registru, který je použit dvěma či více nezávislými instrukcemi, lze eliminovat hazardy WAR a WAW. Přejmenování může probíhat i v kompilátoru na programové úrovni, čímž lze eliminovat některé hazardy již při překladu programu z vyššího jazyka na úroveň strojových instrukcí, avšak kompilátor může pracovat jen s množinou architekturálních registrů definovaných instrukční sadou. Přejmenování registrů na hardwarové úrovni probíhá z logických registrů, definovaných instrukční sadou, do množiny fyzických registrů, která může být i několikanásobně větší.

Rezervační stanice je jednotka, která přebírá instrukce k vykonání v jedné nebo více konkrétních funkčních jednotkách. Rezervační stanice přijímá instrukce z in-

strukční fronty, provádí přejmenování registrů, kontroluje dostupnost operandů a datové závislosti mezi instrukcemi a sleduje stav jednotlivých funkčních jednotek. Na základě všech těchto informací dynamicky rozhoduje, která instrukce se pošle ke zpracování a která funkční jednotka ji bude vykonávat.

Společná datová sběrnice přímo propojuje jednotlivé rezervační stanice a funkční jednotky. Hlavním přínosem společné sběrnice je urychlení přístupu k výsledkům jakékoliv operace, kdy funkční jednotky nemusí čekat na zápis výsledků do registrového pole a poté na vyčtení hodnot operandů. Distribuce dat přes společnou sběrnici tedy umožňuje jednoduše realizovat *forwarding* výsledků a redukuje potřebnou náročnost procesů čtení a zápisů v registrovém poli. Dovoluje také přenechat řízení funkčních jednotek včetně detekce hazardů a vykonávání instrukcí samotným rezervačním stanicím a řízení linky je jednodušší než v případě jedné komplexní jednotky pro ovládání linky.

Každá instrukce prochází třemi kroky zpracování. Těmi jsou vydání instrukce ke zpracování, vykonání instrukce a zápis výsledků.

1. **Vydání instrukce**
   Jelikož vydávání instrukcí probíhá v programovém pořadí, dekódované instrukce jsou uchovávány v zásobníku typu FIFO. Ve fázi vydávání se vezme instrukce z počátku instrukční fronty, zkontroluje se, zda-li je k dispozici příslušná rezervační stanice, a přečtou se operandy, pokud jsou k dispozici v registrovém poli. Pokud rezervační stanice není k dispozici, dochází ke strukturálnímu hazardu a pozastavuje se vydávání instrukcí. Není-li k dispozici některý z operandů, uchovává se informace o tom, která z rezervačních stanic produkuje potřebný výsledek. Dále se v tomto kroku provádí přejmenování registrů.
2. **Vykonání instrukce**
   V tomto kroku se kontroluje, zda-li jsou pro instrukci k dispozici všechny operandy. Chybí-li některý z operandů, kontrolují se data na společné sběrnici. Jakmile jsou operandy k dispozici, předají se rezervační stanici, která poté pošle instrukci ke zpracování do příslušné funkční jednotky. Vykonávání instrukcí, které nemají připravené všechny operandy, je pozastaveno až do doby dokud nejsou k dispozici, čímž se eliminuje RAW hazard. V jednom taktu může být k dispozici i více instrukcí a funkční jednotka pak volí, kterou instrukci vykoná. Pořadí vykonání běžných instrukcí může být libovolné, vykonávání instrukcí LOAD a STORE pro práci s pamětí však přináší další komplexitu. Vykonání těchto instrukcí se skládá ze dvou hlavních kroků - výpočtu adresy a umístění instrukce do paměťové fronty. Instrukce pro načítání mohou být z fronty vykonány okamžitě, jakmile je k dispozici adresa a je dostupná funkční jednotka pro práci s pamětí. Instrukce pro ukládání ale mohou být provedeny až ve chvíli, kdy je známa také hodnota, která se má zapsat do paměti. Pro eli-

minaci dalších rizik, především paměťových hazardů, se vykonávají instrukce LOAD a STORE v programovém pořadí.

Další komplexitu do plánování instrukcí přináší instrukce pro větvení v programu. Vykonávání instrukcí, které předchází instrukce pro větvení, se pozastavuje až do chvíle, kdy je znám výsledek podmínky pro větvení. Tímto se předchází vzniku výjimek odchýlením se od sémantiky v programu, avšak může docházet k prostojům v lince. K eliminaci těchto prostojů se používá spekulací při vykonávání, kdy se povolí vykonání instrukce, avšak se stále čeká na výsledek podmínky pro větvení a výsledek se uchová jen v případě, kdy instrukce měla být vykonána.

3. **Zápis výsledku**

   Jakmile má funkční jednotka připraven výsledek operace, data se vystaví na společnou sběrnici. Ze sběrnice data putují k zápisu do registrového pole a také případně do rezervačních stanic, které čekají na tuto hodnotu.

Implementace algoritmu

Důležitým rozdílem Tomasulova algoritmu vůči metodě Scoreboarding je přístup operandům. Ve chvíli vydání instrukce ke zpracování se přečtou operandy, které jsou dostupné v registrovém poli. Tyto operandy se předají rezervační stanici, která se v okamžiku přejmenování oprošťuje od názvů registrů a adres v registrovém poli. Instrukce již přijde s potřebnými operandy a pokud ty nebyly ve chvíli vydání k dispozici, adresa pro získání hodnoty již nereferuje jméno registru, ale adresu rezervační stanice produkující potřebný výsledek. U algoritmu Scoreboarding naopak instrukce, která nemá operand k dispozici v registrovém poli, čeká na zapsání výsledku do správného registru.

Každá rezervační stanice se skládá z tabulky, kde každý záznam je složen ze sedmi polí. Pole *Op* nese kód operace k provedení. Pole *Qj* a *Qk* nesou adresu rezervační stanice produkující potřebné hodnoty operandů *Vj*, respektive *Vk*. Pokud je v poli Q hodnota '0', tak zdrojový operand je již k dispozici anebo není vyžadován pro danou operaci. Pro každý operand platí pravidlo, že pouze jedno z polí Q a V je platné - pole V nesoucí hodnotu operandu je platné, pouze pokud je pole Q v hodnotě '0'. Následuje pole *A* určené pro instrukce s přístupem do paměti. Uchovává se zde informace pro výpočet adresy a po dokončení výpočtu se zde uchovává samotná adresa pro přístup do paměti. Poslední pole *Busy* poskytuje informaci o stavu rezervační stanice a příslušných funkčních jednotkách. Pro účely přejmenování se také přidává jeden záznam k hlavnímu registrovému poli *Qi*, který nese pro každý registr informaci, která z rezervačních stanic produkuje výsledek k zápisu do tohoto registru. Je-li pole pro konkrétní registr prázdné (nese hodnotu '0'), žádná z aktuálních

instrukcí v lince do něj nezapisuje výsledek a hodnota uchovaná v registru je tedy platná.

Pro větší názornost slouží následující příklad, který ukazuje, jakým způsobem se distribuují informace mezi rezervačními stanicemi při vykonávání instrukcí z fronty. V příkladu se uvažuje celkem pět rezervačních stanic, z nichž dvě slouží pro celočíselný součet (*ADD1* a *ADD2*), jedna pro násobení (*MUL*) a jedna pro každou paměťovou operaci (*LOAD* a *STORE*). Tabulka 2.2 zobrazuje frontu instrukcí, kdy u každé je zobrazen stav, v jakém kroku zpracování se nachází. První instrukce ve frontě, kterou je čtení dat z paměti, již byla vykonána a výsledek operace již byl zapsán do registrového pole. Následující instrukce pro celočíselný součet již byla také vykonána a zapsána. Další instrukce pro načtení dat z paměti je ve stavu rozpracování. Další instrukce pro celočíselný součet nebyla omezena žádnou závislostí, proto mohla být vykonána a postupuje do fáze zápisu výsledku. Pátá instrukce, celočíselné násobení, čeká ke zpracování, jelikož ještě nemá připraven jeden z operandů. Šestá instrukce taktéž čeká na zpracování z důvodu nedostupného operandu. Obě tyto instrukce musí čekat na vyřešení RAW hazardu, než může být vyčtena hodnota operandu a instrukce může být vykonána. Poslední instrukce, jíž je zápis hodnoty z registru do paměti, ještě nebyla vydána.

Tab. 2.2: Příklad fronty instrukcí s vyznačeným stavem rozpracování každé instrukce

Instrukce	Vydána	Zpracována	Výsledek zapsán
ld r1, 0 (r5)	✓	✓	✓
add r2, r2, r1	✓	✓	✓
ld r3, 8 (r6)	✓		
add r7, r4, r8	✓	✓	
mul r1, r2, r3	✓		
add r1, r1, r3	✓		
st r7, 16 (r0)			

Tabulka 2.3 ukazuje status registrového pole pro tuto konkrétní frontu. Jelikož již byl výsledek druhé instrukce zapsán a žádná z aktuálních instrukcí v lince znovu nezapisuje na stejnou adresu, pole pro registr r2 je prázdné a obsahuje validní hodnotu. Naopak do registru r1 zapisují celkem tři instrukce, pole Qi tedy nese adresu rezervační stanice, která produkuje poslední výsledek k zapsání na tuto adresu.

Další tabulka 2.4 zobrazuje stav jednotlivých rezervačních stanic se záznamem

Tab. 2.3: Příklad záznamů v poli Qi pro celočíselné registrové pole

Adresa registru	r0	r1	r2	r3	r4	r5	r6	r7	r8	...
Qi		ADD1		LOAD				ADD2		...

poslední vydané instrukce. Pro znázornění hodnoty registru v poli *V* je použita zkratka *reg(x)*, přičemž písmeno x reprezentuje adresu, ze které se hodnota operandu získává. Jelikož se instrukce vydávají v programovém pořadí, první instrukce *add* byla vydána do rezervační stanice *ADD1* a další instrukce pro součet byla vydána do následující stanice *ADD2*. Po dokončení první instrukce *add* se vydala další instrukce pro součet opět do první stanice. Stanice ADD1 tedy obsahuje instrukci, která nemá k dispozici žádný z operandů a instrukce čeká, než budou výsledky ze stanic MUL a LOAD k dispozici. Instrukce ve stanici ADD2 má naopak oba operandy k dispozici a proto vykonání mohlo začít okamžitě. Instrukce pro násobení ve stanici MUL má k dispozici jen jeden ze dvou operandů, také musí počkat, než bude operand k dispozici ze stanice LOAD. Stanice LOAD obsahuje instrukci pro načtení dat z paměti. Pole Q i V jsou tedy prázdná a pouze pole A nese hodnotu adresy v paměti. Instrukce pro ukládání dat do paměti nebyla vydána a tudíž je stanice STORE prázdná a pomocí pole *Busy* signalizuje, že momentálně nezpracovává žádnou instrukci.

Tab. 2.4: Příklad záznamů v rezervačních stanicích

Rezervační stanice	Op	Qj	Qk	Vj	Vk	A	Busy
ADD1	add	MUL	LOAD				ano
ADD2	add			reg(r4)	reg(r8)		ano
MUL	mul		LOAD	reg(r2)			ano
LOAD	ld					reg(r6) + 8	ano
STORE							ne

Pokud budeme uvažovat situaci, kdy se stav žádné z rozpracovaných instrukcí nezmění a pouze se vydá poslední instrukce z fronty ke zpracování, přibude nám v tabulce záznam pro rezervační stanici STORE. Tato instrukce zanese zápis do pole A, kam se uloží informace potřebné pro výpočet adresy. Instrukce také potřebuje hodnotu dat k zápisu do paměti jako druhý operand. Jelikož výsledek instrukce, která produkuje hodnotu registru r7, ještě nebyl zapsán, zanese se adresa rezervační stanice ADD2 do pole Qk. Jelikož stanice STORE před touto instrukcí nepracovala, změní se i signalizace stavu v poli *Busy* z hodnoty ’0’ (ne) na ’1’ (ano). V tabulce stavu registrového pole nedojde ke změně, jelikož instrukce pro ukládání dat do paměti neprovádí zápis do registrového pole.

Tab. 2.5: Příklad nového záznamu v rezervační stanici STORE

Rezervační stanice	Op	Qj	Qk	Vj	Vk	A	Busy
STORE	st		ADD2			reg(r0) + 16	ano

2.5 Vydávání více instrukcí ke zpracování

Všechny doposud diskutované techniky pro paralelní zpracování instrukcí cílí na dosažení ideálního výkonu, kdy je počet cyklů potřebných k vykonání jedné instrukce (CPI) roven jedné. Pro další navýšení výkonu je nutné dosáhnout počtu CPI menšího než jedna. Zřetězená linka, která vydává každý takt jednu instrukci ke zpracování však nemůže přesáhnout hranici jednoho cyklu na instrukci. Cílem této techniky je umožnit vydání dvou nebo více instrukcí k vykonání za takt. Existují tři základní principy, jak dosáhnout vydání více instrukcí: procesory typu VLIW, superskalární procesory se statickým plánováním a superskalární procesory s dynamickým plánováním,

2.5.1 Procesory VLIW

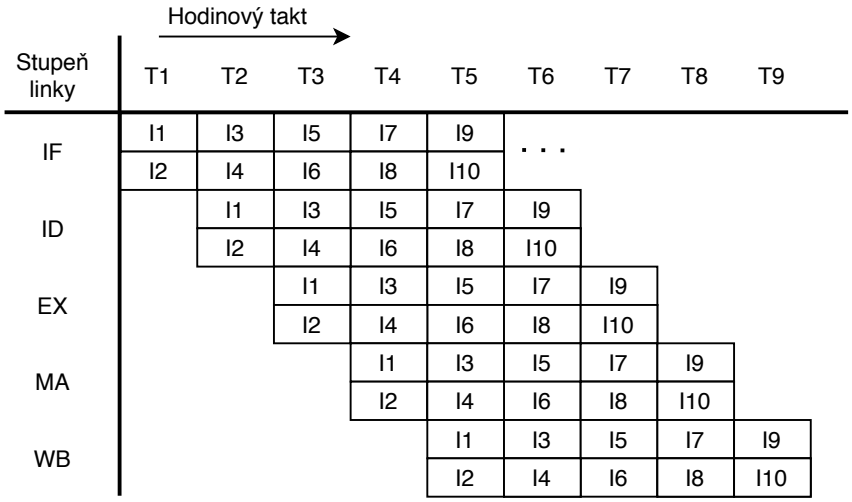
Instrukční sada typu VLIW přináší operace s velmi dlouhým instrukčním slovem. Paralelismu na úrovni instrukcí je dosaženo kódováním více operací do jedné velké instrukce. Délka instrukce dosahuje až stovek bitů a všechny instrukce mají stejnou, pevně určenou délku. Procesor s architekturou VLIW disponuje více funkčními jednotkami a plánování jednotlivých operací probíhá staticky již při překladu programu. Jednotlivé operace v každé instrukci se provádí souběžně, operace tudíž musí být vzájemně nezávislé. Řízení zřetězené linky je tedy jednodušší, protože procesor nemusí vyhodnocovat vzájemné závislosti operací v jedné instrukci. Plánování instrukcí již při překladu však klade vysoké nároky na kompilátor a optimálně napsaný kód programu. Efektivita architektury VLIW vůči superskalárním procesorům roste s rozšiřující se velikostí instrukcí a tudíž počtu paralelně vykonaných operací.

2.5.2 Superskalární procesory se statickým plánováním

Architektura se statickým plánováním se vyznačuje vykonáváním instrukcí v programovém pořadí. Množství vydaných instrukcí za takt není na rozdíl od architektury VLIW pevně určeno, při řešení různých konfliktů a závislostí ve zřetězené lince je možno omezit počet vydaných instrukcí.

Statické plánování je realizováno tak, že z počátku instrukční fronty se nahlédne na daný počet instrukcí a zkontrolují se vzájemné závislosti. Vydávání instrukcí je také omezeno počtem funkčních jednotek. Nedochází-li k žádným konfliktům mezi

instrukcemi, dochází k vydání plného počtu daných instrukcí. Datové, strukturální i řídící hazardy naopak mohou omezit počet vydaných instrukcí. Jelikož vykonávání instrukcí probíhá v programovém pořadí, kontrola vzájemných závislostí a řízení linky jsou poměrně jednoduché. Vzájemné závislosti mezi instrukcemi však brání dosažení ideálního CPI.



Obr. 2.3: Zpracování instrukcí v jednoduché superskalární zřetězené lince

V kontrastu se zpracováním instrukcí v jednoduché řetězené lince vycházející z procesoru DLX (obr.2.2), jehož hodnota CPI v ideálním případě dosahuje jedné, je na obrázku 2.3 principiálně znázorněno ideální zpracování instrukcí v superskalární zřetězené lince složené ze stejných stupňů, jako linka v procesoru DLX. Tato ideální superskalární zřetězená linka, jenž ale nereflektuje přítomnost vzájemných závislostí mezi instrukcemi, vydává ke zpracování každý takt dvě instrukce a může dosáhnout až poloviního CPI oproti skalární lince procesoru DLX.

2.5.3 Superskalární procesory s dynamickým plánováním

Architektura s dynamickým plánováním instrukcí může být založena na libovolném plánovacím algoritmu, nejběžnější je však realizace vycházející z Tomasulova algoritmu. Instrukce jsou vydávány ke zpracování v programovém pořadí. Vydávání instrukcí mimo pořadí by z důvodu četných závislostí vedlo k velkému nárůstu náročnosti procesu vydávání a potenciálně i k nedodržení sémantiky programu. Vykonávání instrukcí již probíhá mimo programové pořadí, obvykle se implementace kombinuje se spekulativním vykonáváním instrukcí. Pro dosažení maximálního potenciálu dynamického plánování se umožňuje vydání všech existujících kombinací daného počtu instrukcí.

Kvůli vzájemným závislostem mezi instrukcemi není vydání většího počtu instrukcí v jednom taktu pro dynamické plánování jednoduchým procesem. Základním předpokladem je, že záznamy v tabulkách jsou v jednom taktu aktualizovány pro všechny paralelně vydané instrukce. První možností je zkrátit proces vydání instrukce na polovinu taktu a tudíž za jeden takt provést tento proces dvakrát pro vydání dvou instrukcí. Tento přístup lze však jen velmi obtížně rozšířit pro vydání tří a více instrukcí v jednom taktu. Další možností je rozšířit jednotku pro vydávání instrukcí tak, aby byla schopna zanalyzovat všechny možné závislosti mezi instrukcemi a zanést záznamy pro všechny vydané instrukce zároveň. Moderní procesory, které vydávají větší množství instrukcí, kombinují z důvodu neúměrného nárůstu komplexity oba tyto přístupy.

Implementaci procesu vydávání více instrukcí lze rozdělit do několika dílčích kroků:

1. Kontrola dostupných rezervačních stanic, zda-li je dostupný dostatek stanic pro vydání libovolné kombinace daného počtu instrukcí. Tato kontrola se provádí dříve, než je znám typ operací a vyžadovaných funkčních jednotek, a to kvůli zjednodušení celého procesu. Počet vydaných instrukcí nemusí dosahovat vždy maximálního počtu, proces lze pro další zjednodušení omezit např. na skupiny instrukcí, které je možné vydat zároveň, a pokud na počátku fronty nejsou zastoupeny instrukce z každé skupiny, vydá se menší počet instrukcí. Také pokud není k dispozici dostatek rezervačních stanic, k čemuž může dojít například v situaci, kdy všechny instrukce ve frontě spadají do stejné skupiny, vydá se omezený počet instrukcí.
2. Analýza všech vzájemných závislostí mezi instrukcemi, které budou vydány. Tento krok nabývá na komplexitě exponenciálně se zvyšujícím se počtem analyzovaných instrukcí.
3. Aktualizace záznamů v tabulkách pro řízení linky a zanesení nových záznamů do příslušných rezervačních stanic. Záznamy jsou vytvořeny na základě výsledků předchozího kroku, kdy musí být správně reflektovány všechny závislosti mezi instrukcemi, aby nedocházelo k odchýlení od sémantiky vykonávaného programu.

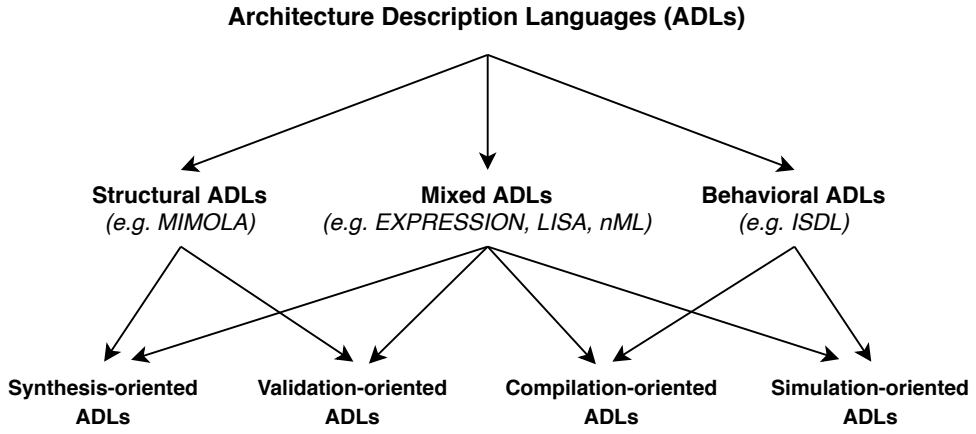
Všechny tyto kroky jsou provedeny v jediném taktu a jsou ukázkou netriviálního procesu vydávání instrukcí v dynamicky plánovaném procesoru. Proces vydávání instrukcí je právě z důvodu exponenciálního nárůstu složitosti při vydávání většího množství instrukcí v moderních procesorech označován jako jeden z hlavních prvků limitující výkonnost celého systému.

3 Vývojové prostředky

K návrhu modelu je použito vývojové prostředí Codasip Studio verze 8.3 od společnosti Codasip, které obsahuje sadu nástrojů pro vývoj mikroprocesorů a práci v jazyce CodAL. Následující kapitoly popisují zařazení jazyka CodAL, základní rysy a možnosti tohoto jazyka a popis vývojového prostředí.

3.1 Zařazení jazyka CodAL

Pro návrh hardwaru se nabízí dvě hlavní skupiny jazyků - HDL a ADL. Jazyky ze skupiny HDL (angl. Hardware description language) slouží pro popis a simulaci hardwaru na RTL úrovni. HDL jazyky, mezi které se řadí především jazyky VHDL či Verilog, však neposkytují optimální prostředky pro popis složitých obvodů úrovně mikroprocesoru, jelikož nenesou dodatečné informace o instrukční sadě, syntaxi strojových instrukcí apod. Jazyky ADL pro popis architektur (z angl. Architecture description language) byly vyvinuty pro poskytnutí dostatečných prostředků potřebných k úplné reprezentaci určitého systému nebo architektury. [7]



Obr. 3.1: Kategorizace ADL jazyků [7]

Jazyky ADL, jak je znázorněno na obrázku 3.1, lze dělit do tří hlavních kategorií na jazyky popisující strukturu (*Structural ADLs*), jazyky popisující instrukční sadu (*Behavioral ADLs*) a smíšené jazyky (*Mixed ADLs*). Jednotlivé skupiny lze dále kategorizovat podle zaměření na jazyky určené k simulaci, překladu, validaci či syntéze. [7]

1. **Jazyky zaměřené na popis struktury**
   Tato skupina jazyků slouží, podobně jako HDL jazyky, k popisu konkrétní implementace instrukční sady. HDL jazyky umožňují popis na nižší úrovni

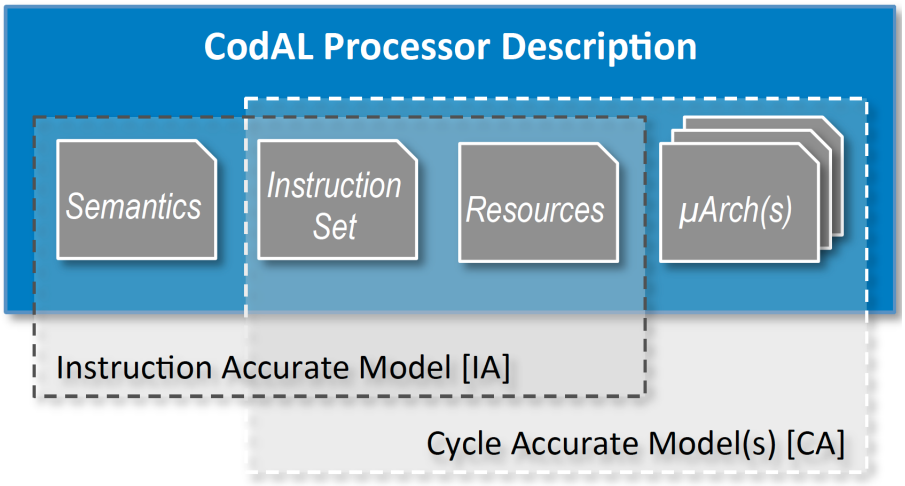
abstrakce RTL, která je dostatečně detailní pro přesný popis struktury a zároveň dostatečně abstraktní pro odstínění detailů nižší hradlové úrovně. Strukturální ADL jazyky umožňují vytvářet popis na různých úrovních abstrakce. Vyšší úroveň abstrakce ADL jazyků umožňuje urychlit vývoj, přičemž nižší úroveň dovoluje přesněji definovat strukturu. Strukturální ADL jazyky jsou vhodné pro syntetézu a simulaci na úrovni hardwaru. Tuto skupinu zastřešuje jazyk MIMOLA.

2. **Jazyky zaměřené na popis instrukční sady**
   Jazyky pro popis instrukční sady procesoru, mezi které se řadí jazyky ISDL nebo Valen-C, umožňují popsat sémantiku jednotlivých instrukcí, nesou informace o všech dostupných prostředcích instrukční sady a dovolují vytvořit model chování. Používají se především ke generování překladačů vyšších programovacích jazyků a dalších nástrojů pro vývoj software. Neobsahují model časování a tudíž nejsou vhodné pro syntézu.
3. **Smíšené jazyky**
   Jazyky se smíšeným zaměřením kombinují vlastnosti obou předchozích skupin. Obsahují prostředky pro popis instrukční sady a zároveň i pro popis struktury modelu procesoru. Hlavními představiteli jsou jazyky EXPRESSION, LISA, nML, ADL++ nebo HMDES.

3.2 Jazyk CodAL

Jazyk CodAL se řadí do skupiny ADL jazyků se smíšeným zaměřením. Poskytuje prostředky pro popis instrukční sady procesoru a jejich zdrojů, sémantiky jednotlivých instrukcí a pro vytvoření modelu procesoru na vyšší úrovni abstrakce. Na obrázku 3.2 jsou znázorněny zdroje jazyka CodAL, pomocí kterých lze vytvořit oba typy modelu procesoru. Prvním typem je model na úrovni instrukční sady IA (angl. Instruction accurate model), druhým model na úrovni cyklů CA (angl. Cycle accurate model). Jazyk CodAL umožňuje vytvořit více různých modelů mikroarchitektur pro jeden společný model na úrovni instrukční sady. [8]

Popis procesoru se v jazyce CodAL dělí na čtyři části taktéž zachycené na obrázku 3.2. První část, sémantika (*Semantics*), popisuje chování jednotlivých instrukcí včetně vyhodnocování výjimek. Instrukční sada (*Instruction Set*) nese informaci o názvech instrukcí, jejich kódování a zápisu v jazyce strojových instrukcí. Další část slouží pro popis zdrojů architektury (*Resources*), jako jsou programový čítač a další registry nebo instrukční a datová rozhraní pro přístup do paměti. Poslední část už slouží pro vytvoření popisu konkrétní mikroarchitektury ($\mu Arch$) a nese informace o časování a konkrétních zdrojích.



Obr. 3.2: Prostředky jazyka CodAL [8]

Základní syntaxe jazyka CodAL, která se využívá například při popisu sémantiky instrukcí, vychází z jazyka ANSI C. Je podporovaná téměř celá množina jazyka ANSI C, s výjimkou ukazatelů. Pro definování instrukční sady slouží prvek *element* zachycený v ukázce kódu 3.1. Prvek *element* může k popisu chování využít i dalších definovaných prvků typu *element* a vytvářet tak složitější konstrukce, které je dále možno seskupovat do skupin pomocí prvku *set*.

Výpis 3.1: Struktura prvku *element* jazyka CodAL

```
element opc_nop
{
    assembly { "nop" }; // zápis v jazyce symbolických adres
    binary { 0x0 };     // binární kódování
    return { 0x0 };     // definice návratové hodnoty
};

element instrukce_nop
{
    use opc_nop as opc;     // použití elementu opc_nop
    assembly { opc };
    binary { 0:bit[1] opc };
    semantics {...};        // definice chování instrukce
}

set instrukcni_sada = instrukce_nop;
```

Pro popis chování na úrovni instrukční sady i na úrovni cyklů lze využít široké škály datových typů a i zde je syntaxe podobná jazyku C. Základní datové typy, mezi které patří například typy *integer* nebo *unsigned integer*, jsou rozšířeny napří-

klad o prvky *signal*, *register*. Jak je zachyceno v úryvku 3.2, jednotlivým prvkům lze přiřadit přesnou bitovou šířku. Nejzásadnějším rozdílem těchto zdrojů vůči klasickým proměnným je, že prvku *signal* je přiřazena výchozí hodnota nula a prvek *register* uchovává poslední přiřazenou hodnotu. Pro popis chování na obou úrovních je použita konstrukce pro popis událostí *event*, přičemž oba modely obsahují alespoň dvě události. Událost *reset* slouží pro nastavení počátečních hodnot registrů a definici počátečního chování a událost *main*, která již obsahuje popis chování modelu. Události typu *event* mohou aktivovat i jiné události *event*, případně speciální prvek *decoder* sloužící pro dekódování instrukcí.

Výpis 3.2: Struktura prvku *event* jazyka CodAL obsahující definici zdrojů

```
event main
{
    use element;     // použití prvku element
    use event;       // použití prvku event

    // definice lokálních zdrojů
    register bit[1] jmeno_registru;
    signal bit[32] jmeno_signalu;

    semantics
    {
        int i;  // definice proměnných
        uint5 unsigned_integer_pet_bitu;
        ...
    };
};
```

3.3 Codasip Studio

Vývojové prostředí Codasip Studio obsahuje sadu automatizovaných nástrojů určených primárně pro návrh aplikačně specifických procesorů. Vytváření návrhu na vyšší úrovni abstrakce použitím jazyka CodAL přináší výhodu v podobě rychlejšího návrhu modelu procesoru a umožňuje automatizovat některé úlohy tohoto procesu. Nástroj Codasip Studio plně využívá výhod ADL jazyka, kdy umožňuje generovat sadu nástrojů pro vývoj softwaru (SDK) a sadu nástrojů pro vývoj hardwaru (HDK). Nástroj je založen na grafickém vývojovém prostředí Eclipse a lze jej ovládat i pomocí příkazového řádku pro usnadnění automatizace některých procesů. [9] a [10]

Simulátor

Tento nástroj slouží k ladění a ověřování funkce vytvořeného hardwaru a softwaru. Simulátor je k dispozici pro model na úrovni instrukcí i pro model na úrovni cyklů. Simulátor pro oba modely je generován z jejich popisu v jazyce CodAL.

Simulace obou modelů je řízena událostmi reprezentovanými prvky *event*. Simulace začíná vykonáním události *reset* a poté pokračuje vykonáváním události *main* až do ukončení simulace, přičemž jednotlivé události se vykonávají v sekvenčně.

Debugger

Nástroj *Debugger* slouží jako rozšíření pro simulátor a přináší možnost ladit program přímo při simulovaném běhu v modelu procesoru. Ladění je podporováno na třech různých úrovních. Úroveň zdrojového kódu vykoná v každém kroku jeden řádek kódu, ať už se jedná o zdrojový kód ve vyšším jazyce C, nebo o kód v jazyce symbolických adres. Úroveň instrukcí umožňuje provést krok o velikosti jedné instrukce v případě modelu na instrukční úrovni nebo krok o velikosti jednoho hodinového cyklu v případě modelu na úrovni cyklů. Nejnižší úroveň simulace probíhá na úrovni popisu v jazyce CodAL, kdy jeden krok simulace odpovídá jednomu výrazu v jazyce CodAL.

Assembler, Disassembler a Linker

Nástroj *Assembler* překládá instrukce na jejich binární reprezentaci. Překlad probíhá z jazyka symbolických adres a výstupem nástroje jsou objektové soubory. Tyto soubory jsou poté nástrojem *Linker* propojeny do binárního kódu a výstupem nástroje je jeden spustitelný soubor čitelný procesorem.

Opačným postupem k procesu překladu lze dosáhnout za pomocí nástroje *Disassembler* převedení binárního kódu ze spustitelného souboru na čitelný kód reprezentovaný v jazyce symbolických adres.

Překladač jazyka C/C++

Vyvíjení programu na vyšší úrovni je umožněno zahrnutím nástroje překladač. Nástroj vychází z otevřeného aplikačního rámce *LLVM Framework*, který je vybaven podporou jazyků C a C++. Tento nástroj slouží pro překlad programu z jazyků C a C++ do jazyka symbolických adres, který je možno nástroji *Assembler* a *Linker* převést na spustitelný soubor.

Nástroj je rozšířen o pokročilé plánování s podporou architektur typu VLIW a řadu parametrů, pravidel a omezení, která překladači umožňují rozpoznávat a eliminovat např. závislosti a hazardy mezi instrukcemi. Překladač je také doplněn

sadou testů z testovacích sad *gcc-torture* a *LLVM testsuite* a standardní knihovnou jazyka C.

Profiler

Nástroj *Profiler* slouží k poskytnutí detailních informací a statistik dokončené simulace. Zaznamenávají se například informace o počtu vykonaných hodinových cyklů či o vykonaných instrukcí. Výsledné informace vypovídající např. o četnosti jednotlivých instrukcí mohou návrháři pomoci při optimalizaci návrhu na úrovni hardware i software. Výsledky lze zobrazit v textové podobě nebo i ve formě přehledných grafů.

Nástroje pro vývoj hardwaru

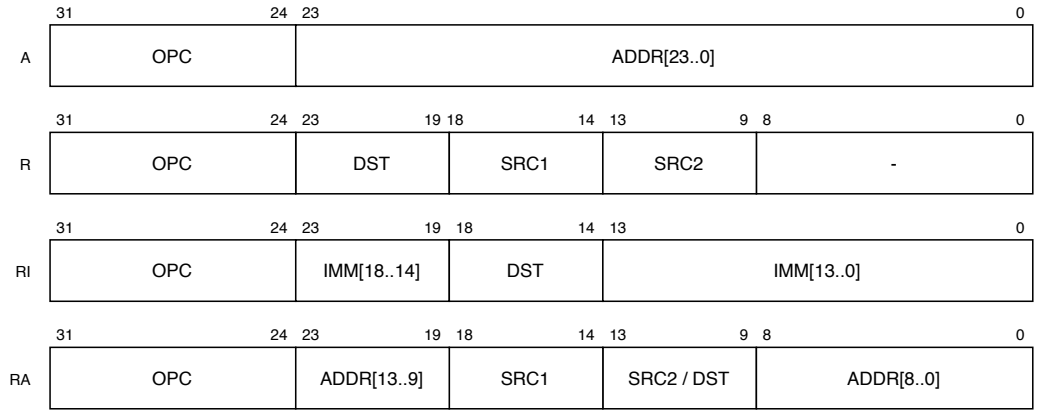
Pro vývoj hardwaru je určeno několik nástrojů, které umožňují generování popisu na úrovni RTL z modelu popsaném v jazyce CodAL, přináší podporu pro simulaci na RTL úrovni, generují verifikační prostředí či poskytují prostředky potřebné pro syntézu na nižší hradlovou úroveň. Nástroj pro vysokoúrovňovou syntézu vytváří z modelu procesoru jeho popis v jednom z jazyků VHDL, Verilog nebo System Verilog. Výsledný popis používá pouze syntetizovatelnou množinu těchto jazyků a používá stejné algoritmy, jako generátor simulátorů, čímž je zaručená vzájemná ekvivalence modelu v jazyce CodAL s modelem úrovni RTL. Pro ověřování konzistence modelů slouží nástroj *Consistency Checker*, který provádí datovou verifikaci nad dvěma modely a vzájemně porovnává výsledky. Společně s vysokoúrovňovou syntézou se generují i podklady a spouštěcí skripty pro některé nástroje určené pro syntézu popisu RTL úrovně na hradlovou úroveň. Generované verifikační prostředí slouží pro funkční verifikaci modelu na úrovni RTL za využití univerzální verifikační metodologie UVM.

4 Codasip uRISC

Mikroprocesor a instrukční sada Codasip μRISC jsou interně vyvíjené ve firmě Codasip a slouží především jako výuková platforma pro tvorbu návodů a zároveň platforma pro demonstraci možností jazyka CodAL a nástroje Codasip Studio. Tento procesor necílí na komplexní funkce a ani na reálnou použitelnost. Základem je zjednodušená instrukční sada a z ní vycházející implementace za využití jazyka CodAL. [11]

4.1 Instrukční sada

Procesory s minimální instrukční sadou (zk. MISC) se vyznačují jednoduchým kódováním instrukcí, které obvykle dovoluje vzniku až 32 instrukcí. Tyto procesory se také vyznačují odlišnou architekturou a šířkou slova osm či šestnáct bitů. Instrukční sada Codasip μRISC se množstvím instrukcí blíží procesorům s minimální instrukční sadou, jelikož obsahuje pouze 35 instrukcí. Kódování instrukcí, které obsahuje dostatek volného prostoru pro až 256 instrukcí, i následná realizace v podobě mikroarchitektury však nesou typické rysy procesorů s redukovanou instrukční sadou. [11]



Obr. 4.1: Formáty instrukcí v sadě Codasip μRISC

Instrukce mají pevně stanovenou velikost na čtyři bajty a pro kódování využívají celkem čtyři různé formáty (obr. 4.1). Instrukční sada definuje pouze základní zdroje, mezi které patří programový čítač a sada registrů pro celočíselné operace. Registrové pole disponuje velikostí 32 položek, pro zakódování adresy jednoho registru je tedy třeba vyčlenit pět bitů. Šířka slova je stanovena na 32 bitů. [11]

Každý formát nese kódování jednotlivých operací v poli s názvem *OPC* o šířce osm bitů. První formát *A* přidává pole pro 24 bitovou adresu a je určen primárně

pro instrukce nepodmíněných skoků. Formát *R* je určen pro standardní aritmetické a logické operace, kódování obsahuje adresy tří registrů, z nichž dva jsou zdrojové operandy *SRC1* a *SRC2* a třetí je cílový registr *DST*. Třetí formát, *RI*, je využit pro načítání konstanty vyjádřené doplňkovým kódem (angl. *signed immediate*) zakódované v poli *IMM* do cílového registru *DST*. Poslední formát, *RA*, vychází z formátu R, obsahuje však adresu pouze jednoho zdrojového operandu a těchto ušetřených pět bitů společně s nevyužitým místem formátu R poskytuje prostor pro 14bitovou adresu vyjádřenou v doplňkovém kódu.

4.1.1 Výčet instrukcí

Pro větší přehlednost jsou instrukce rozděleny celkem do tří základních skupin na instrukce pro práci s pamětí, aritmetické a logické operace a instrukce pro skoky v programu. Tyto skupiny jsou ještě doplněny čtvrtou skupinou speciálních instrukcí, které slouží především pro simulační účely.

1. **Instrukce pro přístup do paměti**
   Instrukce určené pro načítání a ukládání dat používají formát instrukcí RA. Instrukce *load* používají adresu registru *SRC2* jako adresu cílového registru pro načítání dat z paměti, instrukce *store* používají registr *SRC2* jako zdroj dat k uložení do paměti. Obě instrukce používají hodnotu zdrojového registru *SRC1* pro výpočet adresy, k této hodnotě se přičte znaménková konstanta vyjádřená doplňkovým kódem. Tabulka 4.1 obsahuje seznam instrukcí z této skupiny s popisem jejich významu včetně příkladu zápisu každé instrukce v jazyce symbolických adres (assembler).

Tab. 4.1: Instrukce pro práci s pamětí procesoru Codasip µRISC

Jméno instrukce	Zápis v jazyce assembler	Význam operace
ld	r3 = ld [r0 + 4]	Načtení jednoho slova (čtyř bajtů) z paměti
ldhu	r2 = ldhu [r0 + 34]	Načtení dvou bajtů, rozšíření o nuly na šířku slova
ldhs	r2 = ldhs [r0 - 26]	Načtení dvou bajtů, znaménkové rozšíření na šířku slova
ldbu	r2 = ldbu [r2 + 0]	Načtení jednoho bajtu, rozšíření o nuly na šířku slova
ldbs	r2 = ldbs [r0 + 110]	Načtení jednoho bajtu, znaménkové rozšíření na šířku slova
st	st r4 , [r2 + 0]	Uložení jednoho slova (čtyř bajtů) do paměti
sth	sth r2 , [r0 + 26]	Uložení dvou bajtů do paměti
stb	stb r2 , [r0 - 27]	Uložení jednoho bajtu do paměti

2. **Aritmetické a logické operace**
   Aritmetické a logické operace používají formát instrukcí R, s výjimkou instrukce *addi*, jenž používá konstantu zakódovanou ve formátu RA. Instrukce pro porovnání dvou hodnot používají taktéž formát R. Rozdíl mezi dvojicemi instrukcí *slt* a *ult*, respektive *sle* a *ule* je v uvažování znaménkového datového

typu v případě instrukcí začínajících písmenem "s". Dvojice instrukcí *movsi* a *movhi* sloužících pro načtení konstanty do registru využívá formát instrukcí RI. Seznam instrukcí z této skupiny je zobrazen v tabulce 4.2.

Tab. 4.2: Aritmetické a logické instrukce procesoru Codasip μRISC

Jméno instrukce	Zápis v jazyce assembler	Význam operace
add	r2 = add r2 , r3	Součet hodnot dvou registrů
addi	r0 = addi r0 , 16	Součet hodnoty registru a konstanty
sub	r2 = sub r2 , r4	Odčítání hodnoty registru r4 od hodnoty r2
mul	r4 = mul r3 , r2	Násobení hodnot dvou registrů
and	r5 = and r6 , r5	Logická operace AND
or	r4 = or r6 , r10	Logická operace OR
xor	r2 = xor r9 , r2	Logická operace XOR
sll	r3 = sll r9 , r3	Logický bitový posuv vlevo o hodnotu r3
srl	r2 = srl r1 , r2	Logický bitový posuv vpravo o hodnotu r2
sra	r1 = sra r1 , r2	Aritmetický bitový posuv vpravo o hodnotu r2
eq	r3 = eq r3 , r7	Výsledek je jedna, rovnají-li se hodnoty registrů
neq	r4 = neq r24 , r2	Výsledek je jedna, nerovnají-li se hodnoty registrů
slt	r4 = slt r4 , r21	Výsledek je jedna, je-li hodnota r4 menší než hodnota r21
ult	r2 = ult r24 , r9	Výsledek je jedna, je-li hodnota r24 menší než hodnota r9
sle	r2 = sle r2 , r3	Výsledek je jedna, je-li hodnota r2 menší nebo rovna hodnotě r3
ule	r18 = ule r0 , r1	Výsledek je jedna, je-li hodnota r0 menší nebo rovna hodnotě r1
movsi	r0 = movsi 37488	Načtení 19bitové konstanty do cílového registru
movhi	r0 = movhi 1	Načtení 16bitové konstanty do horní poloviny cílového registru

3. **Instrukce pro podmíněné a nepodmíněné skoky**
   Instrukce provádějící nepodmíněné skoky na adresu zakódovanou v konstantě používají formát A. V jazyce symbolických adres lze použít kromě přímého zápisu hodnoty konstanty také jméno návěští, které je cílem skoku a překladač již doplní do kódování instrukce adresu návěští. Druhou variantou jsou nepodmíněné skoky na adresu uloženou v registru, k čemuž je využit formát R. Z tohoto formátu kódování se využije pouze zdrojový registr *SRC2*. Instrukce *call* sloužící k volání podprogramu mají stejnou funkci, jako instrukce *jump*, avšak zapisují návratovou adresu do registru r3, který je pro tento účel vyčleněn instrukční sadou. Instrukce pro podmíněné skoky využívají formátu RA, ze kterého použijí hodnotu zdrojového registru *SRC1* pro vyhodnocení podmínky skoku a konstantu pro výpočet cílové adresy, která je relativní vůči adrese této instrukce. Instrukce pro podmíněné i nepodmíněné skoky jsou shrnuty v tabulce 4.3.
4. **Speciální instrukce**
   Instrukce v této skupině, zobrazeny také v tabulce 4.4, jsou vyčleněny kvůli jejich specifickým funkcím. Instrukce *nop* využívá formátu instrukcí A a nevykonává žádnou funkci. Tato instrukce je použitá například překladačem, který ji může vsunout mezi dvě instrukce zatížené některým konfliktem. Vložení

Tab. 4.3: Instrukce pro podmíněné i nepodmíněné skoky procesoru Codasip μRISC

Jméno instrukce	Zápis v jazyce assembler	Význam operace
jump	jump 16424	Nepodmíněný skok, adresou je konstanta 16424
jump	jump r3	Nepodmíněný skok, adresou je hodnota v registru r3
call	call 38800	Volání podprogramu, adresou je konstanta 38800
call	call r6	Volání podprogramu, adresou je hodnota v registru r6
jumpz	jumpz r18 , -16	Podmíněný skok, provede se je-li hodnota r18 rovna nule
jumpnz	jumpnz r4 , 64	Podmíněný skok, provede se není-li hodnota r4 rovna nule

prázdné operace může být v některých případech přínosné, vede-li to k eliminaci hazardu. Instrukce *halt*, která využívá formát A, slouží jako příkaz pro zastavení simulace. Instrukce *syscall* je kódovaná formátem R, ze kterého využívá pouze pole *SRC1*, ve kterém je pevně zapsána adresa pomocného registru r2. Tato instrukce slouží pro systémová volání, která slouží k interakci s operačním systémem počítače, na kterém je spuštěn simulátor. Systémová volání se v simulátoru používají například k tisknutí textových zpráv do příkazového řádku.

Tab. 4.4: Speciální instrukce procesoru Codasip μRISC

Jméno instrukce	Zápis v jazyce assembler	Význam operace
nop	nop	Operace neprovádí žádnou akci
halt	halt	Příkaz pro simulátor k zastavení simulace
syscall	syscall	Systémová volání

4.1.2 Vytvoření modelu na úrovni instrukční sady

Pro vytvoření modelu na úrovni instrukční sady je nejprve nutné vytvořit popis chování každé instrukce. Toho dosáhneme za použití prvků *element*, obdobně jako v příkladě kódu ukázaném v úryvku 3.1. V popisu sémantiky instrukcí lze také použít některé vestavěné funkce jazyka CodAL, které slouží pro upřesnění různých vlastností instrukcí. Především pro popis chování jednotlivých speciálních instrukcí jsou přichystány funkce *codasip_nop*, *codasip_halt* a *codasip_syscall*. Do sekce sémantiky instrukcí *halt* a *syscall* je také potřeba doplnit další vestavěnou funkci *codasip_compiler_unused*, která zaručí, že překladač tyto instrukce nepoužije při překladů programů z jazyků C a C++. Vytvořením popisu každé instrukce a přiřazením těchto instrukcí do jednoho setu získáme první část modelu - popis instrukční sady. [11] a [12]

Nyní je třeba definovat základní architektonické zdroje. Pro programový čítač slouží klíčové slovo *pc*, které se kombinuje s použitím prvku *register*. Klíčové slovo

je v tomto případě důležité, jelikož identifikuje tento konkrétní typ registru. Pro vytvoření hlavního registrového pole a případně dalších zdrojů instrukční sady slouží klíčové slovo *arch*, jenž se zkombinuje s vybraným zdrojem. Použití klíčového slova je opět stěžejní pro identifikování daného zdroje jako zdroje instrukční sady.

Dalším krokem je definování adresního prostoru a rozhraní pro přístup do paměti. Definice instrukčního a datového rozhraní se provádí v prvku *interface*. Adresní prostor je nastaven prvkem *address_space*. Dalším rysem instrukční sady je definování rozhraní aplikačního rozhraní, který se definuje v prvku *compiler* použitém uvnitř prvku *settings*.

Posledním krokem je vytvoření událostí *reset* a *main*. V události *main* popíšeme, jakým způsobem se načítá a vykonává instrukce. Následující výpis kódu 4.1 ukazuje, jak lze ve třech krocích dosáhnout vytvoření základního modelu na úrovni instrukční sady. V události *main* je použit set pojmenovaný *isa*, který reprezentuje instrukční sadu. Sémantika této události se skládá z načtení instrukce skrze datové rozhraní z adresy programového čítače, inkrementování programového čítače a z vykonání instrukce, jejíž chování je popsáno v sekci sémantiky odpovídajícího elementu.

Výpis 4.1: Událost *main* modelu na úrovni instrukční sady

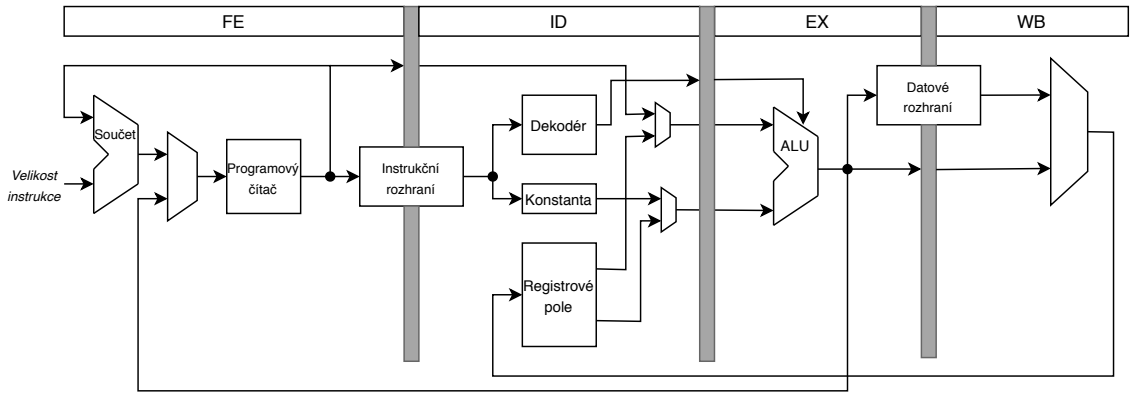
```
event main
{
    use isa;
    register bit[32] r_instruction;

    semantics
    {
        r_instruction = if_fetch[r_pc];
        r_pc += 4;
        isa(r_instruction);
    };
};
```

4.2 Popis mikroarchitektury

Architektura procesoru Codasip µRISC je založena na šířce slova 32 bitů. Procesor je založen na modifikované harvardské architektuře, diskutované v podkapitole 1.2, a je vybaven dvěma samostatnými rozhraními do společné paměti. Instrukční i datové rozhraní mají shodnou šíři adresního i datového kanálu, která je rovna šířce slova, tedy 32 bitům. Pro komunikaci s pamětí je použita sběrnice AHB-3 lite [13].

Zřetězená linka procesoru se skládá celkem ze čtyř stupňů - FE (*fetch*), ID (*instruction decode*), EX (*execute*) a WB (*write-back*). První stupeň linky nazvaný FE se stará o programový čítač a má za úkol posílat požadavky skrze instrukční rozhraní

do paměti, zprostředkovává tedy adresní fázi přenosu po sběrnici. Druhý stupeň (ID) přijímá odpovědi z instrukčního rozhraní, dokončuje tedy přenos datovou fází. Instrukce obdržená z instrukční paměti se dekóduje a přichystají se všechny potřebné operandy. Pro operandy, jejichž hodnoty ještě nejsou zapsány v registrovém poli, jsou zavedeny zkratky z obou následujících stupňů, aby se minimalizovaly prostoje linky. Instrukce dále putuje do dalšího stupně linky, případně, je-li třetí stupeň pozastaven, ukládá se do zásobníku. Třetí stupeň (EX) obsahuje aritmeticko logickou jednotku a slouží k vykonání instrukce. Pro skokové instrukce je zde vypočítána adresa a vyhodnocena podmínka skoku. Pokud instrukce vyžaduje přístup do datové paměti, taktéž se v tomto stupni řeší adresní fáze přenosu po datovém rozhraní. Poslední stupeň linky (WB) slouží k zápisu výsledků do registrového pole a k dokončení datové fáze přenosu do datové paměti. Celkové schéma této zřetězené linky je znázorněno na obrázku 4.2.



Obr. 4.2: Schéma architektury procesoru Codasip μRISC

4.2.1 Vytvoření modelu na úrovni cyklů

Vytvoření modelu na úrovni cyklů je stejně jako v případě modelu na úrovni instrukční sady založeno na událostech *reset* a *main*. Jednotlivé události, nebo například zdroje typu registr, mohou být navíc přiřazeny do konkrétní části linky. Stupně linky se definují za pomoci prvku *pipeline*.

Sdružování událostí a zdrojů do skupiny podle části zřetězené linky přináší výhody snazšího řízení linky. Tato situace je ukázána ve výpisu 4.2, který definuje čtyři stupně procesoru Codasip μRISC a zobrazuje dvě události pro řízení linky. Událost *stall* signalizuje pozastavení linky FE, čímž se zabrání změnám v hodnotách registrů přiřazených do tohoto stupně. Událost *clear* naopak nastavuje všechny zdroje přiřazené do stupně ID na výchozí hodnoty. Další výraznou změnou v modelu na úrovni

cyklů je odlišný přístup k instrukčnímu i datovému rozhraní. Pro tyto účely slouží vestavěné funkce *transport*, jež slouží pro zprostředkování komunikace po sběrnici za využití zvoleného komunikačního protokolu. [11] a [14]

Výpis 4.2: Prvky jazyka CodAL pro řízení linky

```
pipeline pipe { FE, ID, EX, WB };
...
register bit[32] r_id_instruction { pipeline = pipe.ID; };
...
event pipeline_control : pipeline(pipe.FE)
{
    semantics
    {
            pipe.FE.stall();
            pipe.ID.clear();
    };
};
```

5 Vytvoření modelu procesoru

Cílem této práce je diskutovat techniky používané pro zlepšení vlastností zřetězené linky a zvýšení výkonu procesoru paralelním zpracováním instrukcí a přínos vybraných techniky ověřit v praxi. Pro tento účel byla vybrána jednoduchá platforma ve formě instrukční sady a mikroprocesoru Codasip μRISC. Tímto je k dispozici již hotový referenční model jednoduchého procesoru typu RISC, vůči kterému je možno porovnat připravené modely a zhodnotit nejen benefity, ale i limitace zvolených technik.

První zvolenou technikou je metoda vydávání více instrukcí za takt. Model procesoru vznikl rozšířením původní skalární linky procesoru Codasip μRISC na superskalární při vydávání dvou instrukcí ke zpracovaní za jeden hodinový cyklus. Zachován zůstal proces vykonávání instrukcí v programovém pořadí.

Druhou zvolenou technikou je metoda dynamického plánování instrukcí za využití Tomasulova algoritmu. Z důvodů vyhodnocení samostatného přínosu této zvolené metody, ale i z důvodu vyšší náročnosti algoritmu i v tak jednoduché architektuře, jakou disponuje procesor Codasip μRISC, byl tento druhý model založen na skalární lince referenčního modelu, která vydává jednu instrukci každý takt. Cílem je tedy dosáhnout nižšího počtu cyklů na jednu instrukci (CPI), než jakého dosahuje referenční zřetězená linka a více se tak přiblížit ideální zřetězené lince.

5.1 Superskalární procesor se statickým plánováním

Prvním vytvořeným modelem je procesor se superskalární zřetězenou linkou, která je schopna vydat až dvě instrukce v každém hodinovém cyklu. Linka je vybavena statickým plánováním instrukcí a tudíž vykonávání instrukcí i zápis výsledků probíhají v programovém pořadí.

5.1.1 Model na úrovni instrukční sady

Jako základ modelu procesoru na úrovni instrukční sady slouží již popsaný model v sekci 4.1.2. Pro účely generování nástrojů překladače, a nástrojů *Assembler*, *Disassembler* a *Linker* by nebylo nutné model dále upravovat a vygenerované nástroje plně stačí pro návrh a překlad programů. Pro simulační účely je však třeba model rozšířit o několik prvků.

První změna se odehrává na instrukčním rozhraní, které rozšíří datový kanál ze 32 na 64 bitů. Tímto krokem je dosažen nárůst propustnosti instrukčního rozhraní na dvě instrukce o velikosti čtyři bajty (32 bitů). Rozšíření datového kanálu instrukční

sběrnice na osm bajtů však přináší změnu v adresování, protože dle protokolu sběrnice AHB3-lite musí být adresa vždy zarovnána na osm bajtů a nezarovnaný přístup není povolen. Tento problém se týká především skokových instrukcí, které provádí skoky na adresy zarovnané pouze na čtyři bajty. Popis chování je tedy rozšířen o zarovnání adresy programového čítače na osm bajtů a následného výběru správného slova o velikosti čtyř bajtů z osmi bajtů dat, které jsou doručeny datovým kanálem instrukčního rozhraní.

V tuto chvíli je model připraven pro správné generování simulátoru i nástroje *debugger*. Model jsem se ale rozhodl rozšířit o vykonání dvou instrukcí v jednom cyklu simulace. Důvod je zřejmý - model na úrovni instrukční sady reprezentuje chování ideální zřetězené linky a tedy pro následné vyhodnocení výkonu reálné linky představuje model na úrovni instrukční sady maximální dosažitelné CPI a tedy i maximální možný výkon procesoru. Aby byl více zřejmý vliv datových hazardů ve zřetězené lince, model na úrovni instrukční sady je taktéž postižen pokutami za řídící hazardy, čili za vykonání skokových instrukcí. Model tedy vždy načte balík dvou instrukcí, přičemž vykoná první instrukci a nedochází-li k řídícímu hazardu, vykoná i druhou instrukci. Pokuta za vznik řídícího hazardu je tedy jeden takt, respektive ztráta vykonání jedné instrukce. Počet CPI tohoto modelu se tedy nedostane k hranici půl cyklu na instrukci, ale bude mírně větší v závislosti na počtu skokových instrukcí v programu.

Výpis 5.1: Modifikace modelu na úrovni instrukční sady

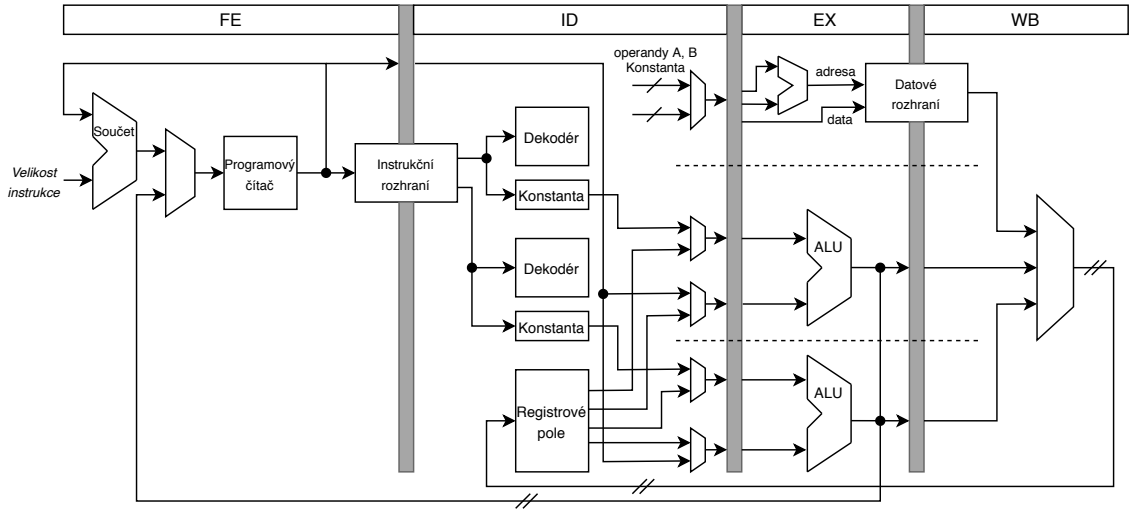
```
addr = r_pc & ~7;           // adresa zarovnaná na 8 bajtů
instr_offset = (uint3)r_pc; // poloha instrukce ve slově

ir1 = if_fetch.read(addr, 8); // načtení 8 bajtů
if (instr_offset[2..2])
    ir2 = if_fetch.read(addr + 8, 8); // načtení 8 bajtů
else
    ir2 = 0;

r_instruction_buffer = (ir2 :: ir1) >> (instr_offset * LAU_W);

r_pc += (INSTR_SIZE);   // inkrementace programového čítače

isa(r_instruction_buffer[31..0]);       // vykonání 1. instrukce
if(!jump(r_instruction_buffer[31..24]))
{
    r_pc += (INSTR_SIZE);
    isa(r_instruction_buffer[63..32]);  // vykonání 2. instrukce
}
```

V úryvku kódu 5.1 je ukázána část události *main* modelu na úrovni instrukční

sady rozšířená o načítání balíku dvou instrukcí a o vykonání dvou instrukcí. Načtení dat v tomto modelu probíhá tak, že se vždy načte balík dat o velikosti osm bajtů z paměti. V případě, kdy adresa nebyla zarovnána na osm bajtů se načte i následující slovo z paměti opět o velikosti osm bajtů. Následně se do zásobníku *r_instruction_buffer* o velikosti dvou instrukcí vyberou dvě správné instrukce podle pozice bajtu ("offsetu") v programovém čítači. Vykonání první instrukce se provede vždy, vykonání druhé instrukce je podmíněno tím, že první instrukce není instrukce skoku. K tomu je zavedena pomocná funkce s názvem *jump*, která vyhodnocuje typ instrukce na základě kódování operace v instrukci.

5.1.2 Návrh zřetězené linky

Základem pro model na úrovni cyklů je čtyřstupňová linka referenčního modelu procesoru. Pro rozšíření linky na superskalární bylo třeba provést několik modifikací, především rozšířit instrukční rozhraní, přidat funkční jednotku a navrhnout nové řízení linky a proces plánování instrukcí. Schéma navržené linky je zobrazeno na obrázku 5.1.



Obr. 5.1: Schéma navržené linky superskalárního modelu procesoru

První stupeň FE byl rozšířen o načítání dat o velikosti osm bajtů z instrukčního rozhraní. Dále je třeba ošetřit načítání dat z nezarovnaných adres. Jelikož model na úrovni cyklů nemůže vyslat dva požadavky na instrukční rozhraní, jako je tomu v modelu na úrovni instrukční sady, zpracovává se pouze požadavek na jedno slovo. Pokud tedy adresa není zarovnaná na osm bajtů, budou platná pouze data ve vyšší polovině slova a tedy pouze jedna instrukce. Aby se předešlo ztrátám výkonu z důvodu načtení pouze jedné instrukce, adresa v programovém čítači se v tomto případě

inkrementuje pouze o velikost jedné instrukce. Adresa, která je zarovnána na osm bajtů, se inkrementuje o velikost dvou instrukcí.

Stupeň ID prošel největším množstvím modifikací, aby bylo možno zpracovat dvě instrukce zároveň. Nejprve byla rozšířena část pro zpracování dat z instrukčního rozhraní a zásobník instrukcí. Proces přebírání instrukcí ze sběrnice musí rozpoznat, která část slova je platná a vybrat validní instrukce. Dále i zásobník instrukcí musí být rozšířen o možnost uložit jednu až dvě validní instrukce. Zásobník je využit v případě pozastavení linky pro prevenci ztráty dat. Další části jsou dekodér instrukcí a modul, který připravuje z binární reprezentace instrukce konstanty. Oba tyto moduly nepotřebovaly projít žádnou změnou, jelikož se neprovedla změna v instrukční sadě. Nicméně oba tyto bloky bylo potřeba vložit do linky podruhé, aby byl tento stupeň schopen zpracovat paralelně dvě instrukce. Dalšími změnami ve stupni bylo rozšíření procesu čtení z registrového pole na čtení až čtyř operandů a rozšíření zkratek výsledků pro metodu *forwarding* o dodatečné výsledky z funkčních jednotek. Nakonec následovalo kompletní přepracování řízení linky včetně vytvoření procesu plánování a vydávání dvou instrukcí paralelně. Tento proces je detailněji popsán v následující podkapitole.

Modifikace ve třetím stupni EX spočívají ve vyčlenění samostatné jednotky pro přístup do paměti a v přidání funkční jednotky. Osamostatnění paměťové jednotky umožňuje zpřehlednit řízení procesu vydávání instrukcí, kdy všechny instrukce typu *load* i *store* jsou zpracovány v této jednotce. Paměťová jednotka zároveň není použita dvakrát, opět z důvodu jednoduššího řízení linky. K procesoru je sice možné připojit paměť obsahující dvě datová rozhraní, nicméně řízení linky je složitější v důsledku dodržení modelu paměťové konzistence a takto složité řízení linky se nehodí pro použití v malých a jednodušších procesorech. Jednotka pro aritmetické a logické operace zůstala nezměněna a přidala se do linky druhá jednotka ALU. Řízení stupně EX bylo rozšířeno o vyhodnocení výsledků dvou skoků, které se mohou vyskytnout zároveň v obou jednotkách ALU. Proces vyhodnocení skoků spoléhá na to, že instrukce v první jednotce ALU se nachází v programu před instrukcí v druhé jednotce ALU. Pro zachování sémantiky programu tedy dostane přednost skok v první jednotce ALU.

Poslední stupeň, WB, byl rozšířen o zápis výsledků z přidaných funkčních jednotek. Z důvodu jednoduché režie byly přidány další dva zápisové kanály do registrového pole. Stupeň WB je tedy teoreticky schopen zapsat v jednom taktu až tři výsledky. V praxi k tomu ale nedojde, protože jsou v každém cyklu vydány vždy maximálně dvě instrukce.

5.1.3 Statické plánování instrukcí

Instrukce tohoto procesoru se dají rozdělit do tří skupin. Skupiny vycházejí ze skupin definovaných v sekci 4.1.1. První skupinou jsou tedy aritmetické a logické instrukce (AL). Tyto operace jsou vykonány v jednotce ALU a jsou navíc rozšířeny o operace ze skupiny speciálních instrukcí. Druhá skupina instrukcí slouží pro přístup do paměti (LS), spadají sem tedy instrukce *LOAD* a *STORE*, které jsou zpracovány paměťovou jednotkou. Poslední skupinou jsou instrukce pro změnu programového čítače (JMP), tedy instrukce pro podmíněné a nepodmíněné skoky. Tyto instrukce jsou zpracovány taktéž jednotkou ALU, která provádí výpočet adresy a vykonání instrukce je doplněno o vyhodnocení podmínky skoku.

Tab. 5.1: Závislosti kontrolované při staticky plánovaném procesu vydávání instrukcí

1. instrukce	2. Instrukce	Vzájemné konflikty	Počet vydaných instrukcí
AL	AL	datový	1-2
AL	LS	datový	1-2
AL	JMP	datový	1-2
LS	AL	datový	1-2
LS	LS	datový, strukturální	1
LS	JMP	datový	1-2
JMP	AL	řídící	2
JMP	LS	řídící	2
JMP	JMP	řídící	2

Při návrhu superskalární linky pro vydání dvou instrukcí byla linka vybavena dvěma jednotkami ALU místo jedné. Tím je dosaženo eliminování strukturálních hazardů mezi instrukcemi ze skupiny AL a JMP. Z důvodu jedné paměťové jednotky ale v lince stále dochází ke strukturálním hazardům mezi dvěma instrukcemi ze skupiny LS. Je-li první ve frontě instrukce pro podmíněný nebo nepodmíněný skok, nedochází k žádnému datovému hazardu. V takovém případě je možné vydat ke zpracování obě instrukce. Ovšem pokud je skok proveden, dochází k řídícímu hazardu a výsledek druhé vydané instrukce se již nesmí zapsat. Jedná se tak o velmi jednoduchou formu spekulativního vykonávání druhé instrukce, protože při jejím vydání ještě není známo, zda-li se instrukce má skutečně vykonat.

V případech, kdy je na začátku fronty jiná instrukce než instrukce skoku je vždy možný vznik datového konfliktu vedoucí k RAW hazardu. I když se v této superskalární lince vykonávají instrukce v programovém pořadí, může zde v některých mezních případech dojít i k hazardu WAW, který nastává, zapisují-li obě instrukce v jeden takt výsledek na stejnou adresu v registrovém poli. Poslední typ datového

konfliktu (WAR) v této lince nevzniká. Pro vyřešení vznikajících konfliktů se v aktuálním cyklu vydá se ke zpracování pouze první instrukce. V dalším taktu je možno vydat i druhou instrukci, jelikož v případě hazardu RAW již bude operand připravený v důsledku použití metody *forwarding* v lince a případný hazard typu WAW je vyřešen opožděním zápisu výsledku druhé instrukce.

Vzájemné konflikty vznikající mezi instrukcemi, které jsou závislé na pořadí instrukcí ve frontě, jsou zachyceny v tabulce 5.1. Na základě těchto vzájemných interakcí byl navržen systém vydávání instrukcí ke zpracování. Tabulka také zachycuje, jak z důsledků řešení různých konfliktů klesá počet vydaných instrukcí v jednom cyklu. Dochází-li v lince ke strukturálnímu hazardu, jediné možné řešení v navržené lince je omezení počtu vydaných instrukcí. Datové konflikty mezi souběžně vydanými instrukcemi jsou také řešeny omezením počtu vydaných instrukcí. Pokud však k datovému konfliktu nedochází, vydává se plný počet instrukcí ke zpracování. Řídící konflikty nezpůsobují žádná omezení v procesu vydávání instrukcí. Linka však musí zajistit, že nedojde k porušení sémantiky programu. Výsledek druhé instrukce se tedy nezapíše, pokud první instrukce vykonala skok.

5.1.4 Ladění modelu a chyby v návrhu

Po vytvoření prvotní verze modelu jsem přistoupil k jeho otestování a případnému ladění chyb. I přes skutečnost, že hned první verze modelu nepracovala správně, se neprokázaly žádné konceptuální chyby v návrhu a výsledná implementace je tedy přesným obrazem navržené architektury.

Chyby, které se v prvotním modelu vyskytovaly, se týkaly šířky datového kanálu instrukčního, ale i datového rozhraní. Nástroj Codasip Studio bohužel neumožňuje pro simulační účely vygenerovat společnou paměť, která by měla rozdílné šířky datových kanálů, s čímž jsem při návrhu linky nepočítal. Jednotku pro práci s pamětí tedy bylo nutné rozšířit pro práci s daty o šířce slova osm bajtů, podobně jako bylo navrženo zpracování dat na instrukčním rozhraní. Další chyby vyskytující se v modelu vycházely ze špatného pochopení výstupů dekodéru. V prvotní verzi mohlo tedy dojít k hazardu typu WAW, který způsobil zapsání špatného výsledku. Také jsem si všiml, že instrukce *nop*, která neprovádí žádnou akci, taktéž způsobuje hazardy RAW a WAW v lince, což však nemělo vliv na odchýlení od sémantiky programu a jednalo se tedy problém s čistě výkonnostními dopady. Obě tyto chyby byly opraveny drobným zásahem do výstupních signálů dekodéru. Další drobné chyby v návrhu byly způsobeny např. překlepy při psaní popisu modelu.

Chyby jsem zpočátku ladil na vlastních velmi jednoduchých programech napsaných v jazyce symbolických adres. Cílem těchto programů bylo prozkoumat, jestli se řízení linky chová správně v konfliktních situacích zobrazených v tabulce 5.1 a ta-

kové programy typicky obsahovaly méně než deset instrukcí včetně instrukce *halt* pro ukončení simulace. Další ladění jsem prováděl na programech z testovacích sad přiložených k nástroji překladač. Tyto programy jsou napsány v jazycích C nebo C++ a poté přeloženy na jazyk symbolických adres. Tento přístup se osvědčil jako výhodný, protože přeložený program obsahuje širokou škálu instrukcí v různých kombinacích, což zaručí ověření řady mezních situací v lince. K ladění jsem používal simulátor společně s nástrojem *Debugger*. Při ladění se osvědčil i nástroj *Consistency Checker*, který porovnává model na úrovní instrukční sady s modelem na úrovni cyklů a dokáže identifikovat například odlišné zápisy do paměti či registrového pole.

5.2 Skalární procesor s dynamickým plánováním

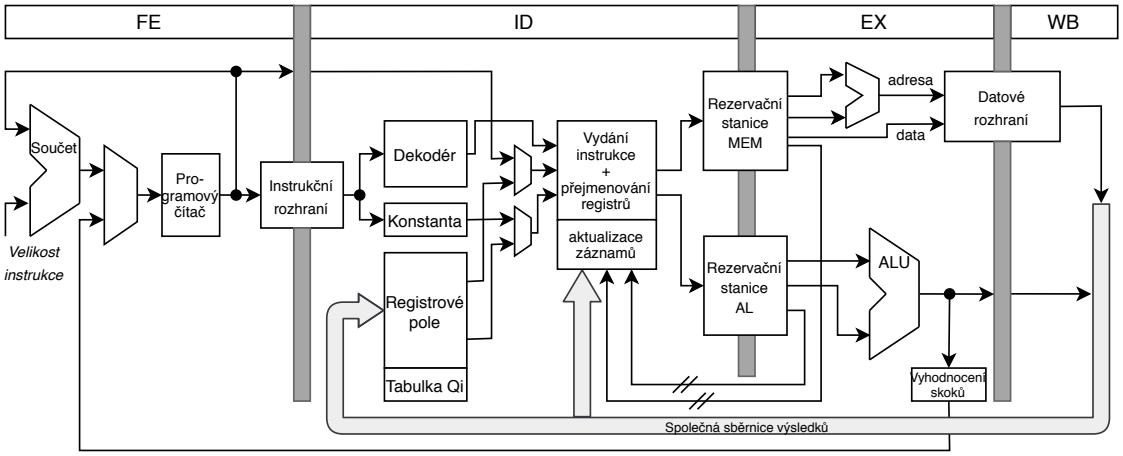
Druhý vytvořený model procesoru je založen na skalární zřetězené lince, která v každém hodinovém cyklu načte a vydá nejvíce jednu instrukci. Pro proces vydávání instrukcí byl zvolen a implementován Tomasulův algoritmus pro dynamické plánování instrukcí. Vydávání instrukcí v implementovaném algoritmu probíhá v programovém pořadí, vykonávání instrukcí již nemusí toto pořadí respektovat.

5.2.1 Model na úrovni instrukční sady

Při návrhu modelu na úrovni instrukční sady nebylo třeba implementovat žádné změny a tento procesor tedy používá referenční model typu IA. Vygenerované nástroje *assembler*, *disassembler*, *linker* i simulátor a *debugger* jsou plně dostačující pro návrh, překlad i ladění programu. V modelu na úrovni instrukční sady však nelze nasimulovat chování navržené zřetězené linky s dynamickým plánováním, která vlivem přejmenování neprovádí všechny zápisy do registrového pole. Použití nástroje *Consistency Checker* je tedy omezené, jelikož nástroj detekuje chybějící zápisy do registrového pole v modelu na úrovni cyklů, což značně ztěžuje proces ladění modelu.

5.2.2 Návrh zřetězené linky

Zřetězená linka pro tento model procesoru byla téměř celá přepracována. Byl zachován stejný počet stupňů linky, tedy čtyři, a původní mechanismus načítání instrukcí skrze instrukční rozhraní. Původní funkční jednotka byla stejně jako v případě předchozího modelu rozdělena na dvě samostatné jednotky, přičemž každá byla vybavena vlastní rezervační stanicí. Zjednodušené schéma navržené linky je zachyceno na obrázku 5.2.



Obr. 5.2: Schéma navržené linky dynamicky plánovaného modelu procesoru

Z původní linky byl zachován celý mechanismus načítání instrukcí, což znamená že první stupeň FE zůstal beze změny a z druhého stupně ID byla zachována část pro zpracování dat společně se zásobníkem instrukcí.

Druhý stupeň zřetězené linky ID si dále ponechal už jen modul pro přípravu konstant z kódované instrukce. Vlivem přepracování celého řízení linky musel projít změnou i dekodér, protože některé původní výstupy byly zbytečné a jiné nenesly potřebné informace. Nový dekodér je jednodušší, protože se společně s instrukcí předává linkou méně informací, z nichž část byla přesunuta do příslušných funkčních jednotek. Nové výstupy dekodéru pomáhají určit typ instrukce podle rozdělení popsaném v sekci 4.1.1. Kromě řídících signálů pro výběr obou operandů a cílového registru byl přidán i signál pro ovládání výběru hodnoty pro pole adresy rezervačních stanic. Z dekódované instrukce je připraven záznam do tabulky rezervační stanice a ten je poté předán modulu pro plánování instrukcí.

Tento modul obdržuje informace z rezervačních stanic a na základě dostupností operandů provádí přejmenování registrů. Poté je připravený záznam zanesen do dostupné rezervační stanice podle typu instrukce. Zároveň se v tomto modulu sbírají výsledky vykonaných instrukcí, jež jsou k dispozici pro aktuálně vydávanou instrukci a také pro aktualizaci všech záznamů v rezervačních stanicích. V neposlední řadě zde také probíhá aktualizování záznamů v tabulce Qi, která obsahuje záznamy o čekajících zápisech výsledků do registrového pole. Detailněji je tento proces nastíněn v následující podkapitole.

Stupeň EX byl doplněn o proces čtení tabulky rezervačních stanic. Tento proces je určen ke třem účelům. Nejprve se analyzují pole *Busy*, *Qj* a *Qk*. Z těchto polí se získává informace o tom, zda-li se v rezervační stanici nachází instrukce, která má dostupné oba operandy a je tedy možné ji vykonat. Dále se zde také určuje, která

stanice je k dispozici pro zapsání nového záznamu. Oba tyto procesy jsou implementovány za pomoci jednoduchého vyhledávacího algoritmu, který vždy začíná od prvního záznamu a zvolí se první nalezená shoda. Tento způsob není ideální, avšak je jednodušší na implementaci. Pokud je nalezena instrukce, která je připravena k vykonání, je vyzvednuta a poslána ke zpracování do funkční jednotky a zároveň se pošle instrukce k vymazání záznamu této vydané instrukce. Funkční jednotky tuto instrukci zpracují a posílají výsledky do dalšího stupně.

Poslední stupeň WB zpracovává datovou fázi přenosu skrze datové rozhraní a zapisuje případné výsledky do registrového pole. Přidána zde byla společná sběrnice, na které jsou vystaveny výsledky operací včetně adresy rezervační stanice, ze které byl výsledek vyprodukován. Tato sběrnice je zavedena napříč linkou zpět do stupně ID, kde slouží pro aktualizaci záznamů rezervačních stanic a také aktualizaci tabulky registrového pole *Qi*.

5.2.3 Dynamické plánování instrukcí

Algoritmus plánování instrukcí byl navržen podle chování popsaného v teorii v sekci 2.4.4. V lince byly připraveny dvě rezervační stanice, přičemž každá je navržena jako tabulka o daném počtu záznamů. Větší počet záznamů znamená zvýšení náročnosti na řízení linky, příliš málo záznamů však snižuje efektivitu algoritmu. Počet záznamů byl při návrhu s přihlédnutím k těmto faktorům experimentálně zvolen na čtyři. Předpokladem bylo, že tento počet záznamů je ve čtyř stupňové zřetězené lince s instrukcemi s velmi malou latencí dostatečně velký pro projevení benefitů dynamického plánování a zároveň je tento počet i dostatečně malý, aby bylo možné jej realizovat co nejjednodušeji.

Rezervační stanice byly pojmenovány MEM - v případě stanice pro paměťovou funkční jednotku - a AL v případě stanice pro aritmeticko logickou jednotku. Tabulka registrového pole zůstala pojmenována Qi. Všechny tři tabulky byly implementovány jako registrová pole. Tabulka Qi má šířku jednoho pole odpovídající čtyřem bitům, které jsou potřeba pro adresování osmi rezervačních stanic, přičemž nulová hodnota je rezervována pro signalizaci stavů, kdy není k dispozici žádný validní výsledek (použito na sběrnici výsledků) nebo kdy je připraven operand (použito v polích Qj a Qk rezervačních stanic). Počet záznamů v tabulce Qi je roven počtu registrů definovaným instrukční sadou. Tabulky pro stanice MEM a AL obsahují stejná pole jako tabulky z teoretické přípravy, jedná se tedy o pole *Op*, *Qj*, *Qk*, *Vj*, *Vk*, *A* a *Busy*. Tato pole byla doplněna dalším polem nazvaným *Dst*, které uchovává adresu cílového registru. Tato adresa je využita ve stupni WB při zápisů výsledků do registrového pole, aby se předešlo nutnosti implementovat vyhledávací algoritmus pro tabulku Qi. Rozdíly mezi oběma rezervačními stanicemi jsou pouze v šíři pole *Op* nesoucím

kódování operace. V případě tabulky MEM má kódování operace šířku čtyři bity a celý záznam má velikost 105 bitů. Kódování operací v tabulce AL má šířku devět bitů a velikost záznamu je tedy 110 bitů.

Proces vydávání instrukce probíhá v několika krocích. U dekódované instrukce se zkontroluje dostupnost obou operandů. Kontroluje se jejich dostupnost v registrovém poli, pokud však příslušná hodnota v poli *Qi* není rovna nule, operand není k dispozici a potřebný výsledek je produkován ze stanice s adresou uloženou v tomto poli. Dalším krokem je kontrola sběrnice výsledků, zda-li není k dispozici hodnota chybějícího operandu. Pokud je operand k dispozici, zapíše se jeho hodnota do příslušného pole *Vj* nebo *Vk*, v opačném případě se zapisuje adresa stanice, která produkuje výsledek do odpovídajícího pole *Q*. Z výstupů dekodéru je každé instrukci také přiřazen cílový registr a hodnota, která nese informaci, zda-li daná instrukce zapisuje do cílového registru. Pokud instrukce vyžaduje zápis cílového registru, je připraven záznam pro tabulku *Qi*. Tímto je dokončen proces přejmenování registrů a další reprezentace instrukce se již odkazuje na výsledky jiných operačních stanic. Po procesu přejmenování je připraven nový záznam pro tabulku rezervační stanice. Dle typu instrukce se kontroluje, je-li dostupná potřebná stanice. Pokud ano, záznam se zanese. V opačném případě se pozastavuje vydávání instrukce a celý proces se opakuje v dalším taktu.

Speciální případy nastávají v lince při vydání instrukce skoku nebo operace ze skupiny speciálních instrukcí. Při vydání skokové instrukce se zastavuje proces vydávání až do chvíle, kdy je znám výsledek skoku. Tímto se nedovoluje žádná forma spekulativního vykonávání instrukcí a zároveň se i eliminuje vznik řídících hazardů. Další speciální situace se týká skupiny speciálních instrukcí, především simulačních instrukcí *halt* a *syscall*. Je-li jedna z těchto instrukcí detekována, pozastavuje se proces vydávání až do chvíle, kdy jsou pole *Busy* u všech záznamů prázdná a tedy všechny instrukce z rezervačních stanic už byly vydány. Tímto je zaručeno zapsání výsledků všech rozpracovaných instrukcí a tedy i správná funkce simulačních instrukcí. Kromě zmíněných řídících hazardů, které jsou eliminovány, může v lince dojít ke strukturálnímu hazardu. Ten nastává v situaci, kdy není místo ve vyžadované funkční jednotce a tato situace, jak bylo zmíněno výše, také vede k pozastavení procesu vydávání. V důsledku použití metody přejmenování v lince nedochází k datovým hazardům, které by bránily vydání instrukce.

Nakonec nastává proces aktualizace záznamů v jednotlivých tabulkách. Společná sběrnice výsledků v tomto modelu obsahuje výsledky až dvou operací. Informace na sběrnici jsou adresa stanice produkující výsledek, hodnota výsledku a adresa cílového registru. Aktualizace tabulky Qi probíhá tak, že se na adrese cílového registru zkontroluje hodnota adresy rezervační stanice. Pokud je tato hodnota nenulová a je rovna adrese na sběrnici, jedná se o dokončení operace, která zapisuje výsledek do

registrového pole a záznam v tabulce Qi je tedy možno vymazat. Obdobně probíhá i proces aktualizace tabulek rezervačních stanic. Náročné na celém procesu aktualizace záznamů je množství prováděných kontrol, protože každý z osmi záznamů se porovnává vůči výsledkům na společné sběrnici.

5.2.4 Ladění

Proces ladění jsem započal, stejně jako v případě předchozího modelu, za využití vlastních jednoduchých programů napsaných v jazyce symbolických adres. Jeden z prvních programů obsahoval přibližně deset stejných instrukcí pro součet za sebou. Tento program skvěle posloužil pro kontrolu aktualizací jednotlivých tabulek, kontrolu předávání výsledků ze společné sběrnice a v neposlední řadě i ověření situace, kdy nastane strukturální hazard vlivem zaplnění rezervační stanice.

Po prvotním odladění chyb v důsledku překlepů při psaní popisu modelu jsem se opět přesunul na ladění na programech napsaných v jazyce C, které pocházejí z testovacích sad překladače. Tyto testy mi pomohly odhalit chyby při řešení řídících hazardů a dalších mezních situací v lince. Jeden závažnější problém se týkal například procesu předávání výsledků ze sběrnice, kdy docházelo k vyčtení nevalidních hodnot a jejich zapsání do rezervačních stanic, což v důsledku k vedlo k nesprávným výpočtům.

Po odladění všech těchto chyb prvotního návrhu se mi podařilo ověřit platnost celého konceptu navrženého v této práci. Dílčí chyby byly způsobeny detailními implementačními problémy a všechny zjištěné chyby se mi podařilo opravit. V tento okamžik byl již model připraven pro poslední změnu v návrhu.

5.2.5 Paměťová konzistence

V prvotní verzi návrhu byla rezervační stanice paměťové jednotky implementována stejným způsobem, jako stanice pro jednotku AL. Vyhledávací algoritmus tedy vždy zvolil první dostupnou instrukci ze stanice a tu předal do funkční jednotky k vykonání. Následující úryvek kódu 5.2 zobrazuje příklad problému, který nastává při tomto přístupu. Tento příklad je vybrán z jednoho programu z testovací sady překladače, který jsem použil při ladění, jde tedy o praktickou ukázku problematiky.

Výpis 5.2: Příklad sekce kritické pro dodržení paměťové konzistence

```
r4 = add r4 , r5
st r4 , [ r0 + 12 ]
r4 = ld [ r0 + 12 ]
```

Vezměme v úvahu, že v daný okamžik je rezervační stanice MEM prázdná, stanice AL obsahuje pouze jeden volný záznam a to poslední záznam na adrese 4 a jsou

dostupné všechny operandy. První instrukce je vydána ke zpracování do stanice AL. Další ve frontě je instrukce *store*, která do paměti ukládá výsledek předchozí operace. Tato instrukce je uložena do stanice MEM jako první záznam, nemá však k dispozici potřebný operand ve formě registru r4. Další instrukce, *load*, načítá data z paměti ze stejné adresy. Instrukce má ale k dispozici všechny operandy a je zapsána jako druhý záznam v tabulce MEM. V následujícím taktu, kdy od vydání první z trojice instrukcí uběhly dva takty, se tedy ze stanice AL bude vydávat instrukce ze třetího záznamu. Hodnota registru r4 tedy ještě není známa. Stanice MEM však při procházení zjistí, že na druhém záznamu se nachází instrukce, kterou je možno vykonat a instrukce *load* je vydána ke zpracování dříve než instrukce *store*. Nastává tedy situace, kdy se načte nesprávná hodnota z paměti, dochází k zápisu nesprávného výsledku do registrového pole a odchýlení od správného vykonání programu.

Aby bylo dosaženo zachování konsistentního paměťového modelu a zajistilo se tedy správné vykonání všech instrukcí pro načítání či ukládání dat, bylo zvoleno řešení, které zajistí vykonání těchto instrukcí v programovém pořadí. Toto řešení přistupuje k tabulce rezervační stanice jako ke kruhovému zásobníku. Pro tyto účely byly zavedeny dva ukazatele. První ukazatel slouží k identifikaci volného záznamu v tabulce. Je využit při procesu vydávání instrukcí a po vydání instrukce do stanice MEM se inkrementuje. Druhý ukazatel slouží pro výběr instrukce k vykonání. Ukazuje na záznam instrukce, která má být vykonána jako následující podle programového pořadí. Funkční jednotka čeká, dokud tato instrukce není připravena. Ve chvíli, kdy jsou dostupné všechny operandy, se instrukce vydá a druhý ukazatel je taktéž inkrementován. Mezní situace, kdy oba ukazatele míří na stejný záznam může znamenat, že je celá tabulka plná a nebo je celá prázdná. Mezi těmito případy se rozhoduje vyhodnocením polí *Busy* v tabulce rezervační stanice, které poskytují informaci o platnosti záznamů.

Zvolené řešení zaručuje vykonání instrukcí pro práci s pamětí v programovém pořadí, čímž je zajištěna konzistence paměťových operací. Z rezervační stanice se tedy stává fronta instrukcí pro práci z pamětí. Toto řešení bylo zvoleno z důvodu jednodušší implementace na rozdíl od plnohodnotných front pro načítání a ukládání dat, které jsou popsány v teoretické části této práce. Nevýhodou zvoleného řešení je nižší efektivita, jelikož není dovoleno vykonání mimo programové pořadí ani těm instrukcím, které nejsou zatíženy žádnou závislostí.

6 Výkonnostní testy

Pro vyhodnocení přínosu implementovaných technik byly vybrány syntetické výkonnostní testy (angl. *benchmark*) pro zjišťování výkonu v celočíselných operacích. Zvolenými výkonnostními testy jsou sady pro testování procesorů pro vestavěné systémy Coremark a Dhrystone. Jelikož cílem této práce je pouze vyhodnotit přínos zvolených implementačních technik, vytvořené modely neprošly plným procesem verifikace a potenciálně tedy mohou stále obsahovat nějaké chyby. Zvolené výkonnostní testy provádí sadu celočíselných operací a na konci běhu programu obsahují kontrolu správnosti výpočtů. Výsledné skóre výkonnostního testu lze tedy prohlásit za validní, pokud je správnost výsledků na konci programu ověřena.

6.1 Vliv optimalizací překladače

Při vyhodnocování výsledků je třeba brát v potaz řadu faktorů. Jedním ze stěžejních vlivů na výsledné skóre ve výkonnostních testech vnáší do měření úroveň optimalizací překladače. Překladač v nástroji Codasip Studio je schopen analyzovat vzájemné závislosti mezi instrukcemi při procesu překladu a například přesouváním některých nezávislých instrukcí eliminovat vznik některých hazardů v lince, čímž dochází k navýšení výkonu zřetězené linky.

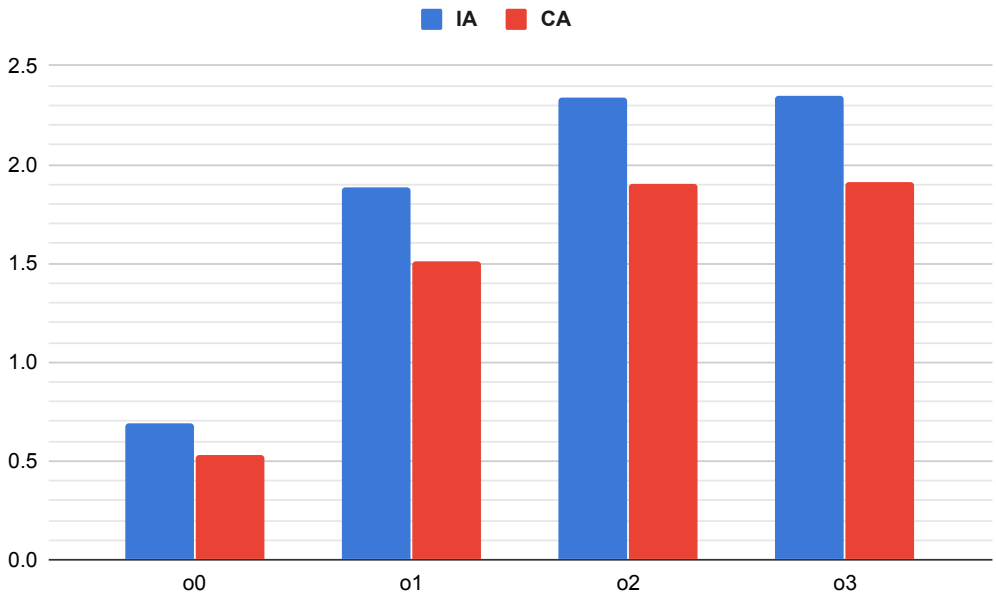
Překladač podporuje řadu parametrů, kterými lze ovlivnit výsledek překladu. Základní parametr sloužící pro kontrolu stupně optimalizace může být aktivován přidáním parametru "-O"mezi parametry překladu. Tento parametr má v současnosti definovaných několik stupňů optimalizace, vyjádřených čísly v rozsahu 0 až 3. Optimalizace na stupni nula znamená, že překladač neprovádí žádné optimalizace, zatímco při překladu na stupni optimalizace tři provádí překladač nejvíce optimalizací kódu. Vyšší stupeň optimalizace může zvýšit výsledný výkon procesoru, avšak za cenu vyšší velikosti výsledného programu a déle trvajícího procesu překladu. Nástroj překladač podporuje ještě několik dalších stupňů optimalizace pro zaměření na minimální velikost kódu či další agresivnější optimalizace pro zvýšení výkonu.

Tab. 6.1: Vliv optimalizací překladače měřený na testu Coremark

Stupeň optimalizace	O0	O1	O2	O3
Dosažené skóre modelu IA [Coremark/MHz]	0,69	1,88	2,34	2,35
Dosažené skóre modelu CA [Coremark/MHz]	0,53	1,51	1,90	1,91

Pro vyhodnocení vlivu překladače byl použit test Coremark, který byl nastaven pouze na délku jedné iterace testu. Takové měření neposkytuje velmi přesné skóre,

ale pro demonstraci vlivu optimalizací překladače je tento příklad dostačující. V tabulce 6.1 jsou ukázány výsledné hodnoty naměřené na referenčním modelu. Měření bylo provedeno na simulátorech modelu na úrovni instrukční sady (IA) i modelu na úrovni cyklů (CA). Z tabulky je patrný obrovský rozdíl mezi stupni optimalizace 0 a 1, přičemž při zapnutí optimalizací dochází k nárůstu výkonu modelu IA o více než 2,7krát, na modelu CA je rozdíl ještě nepatrně větší. Při zapnutí nejvyššího stupně optimalizace překladu narůstá výsledné skóre v obou případech o násobek přibližně 1,25 v porovnání s prvním stupněm optimalizace. Tyto výsledky jsou také graficky znázorněny na obrázku 6.1.



Obr. 6.1: Rozdíly ve skóre testu Coremark způsobené vlivem stupně optimalizace

Vliv optimalizací překladače je tedy značný a to nejen v případě zlepšování výkonu modelu na úrovni cyklů, ale i v případě ideálního simulačního modelu. Výsledky testů je tedy důležité porovnávat na stejném stupni optimalizací pro vytvoření rovnocenných podmínek pro srovnání. Z výsledků referenčního modelu je také patrné, jak velký vliv mají pokuty za vznik hazardů na výsledný výkon zřetězené linky. Porovnáme-li výsledky na nejvyšším stupni optimalizace, model na úrovni instrukční sady, který dosahuje ideálního výkonu dané architektury, dosahuje skóre 2,35 Coremark/MHz, zatímco reálný model zřetězené linky dosahuje skóre 1,91 odpovídajícímu pouze přibližně 81.2 % ideálního výkonu procesoru.

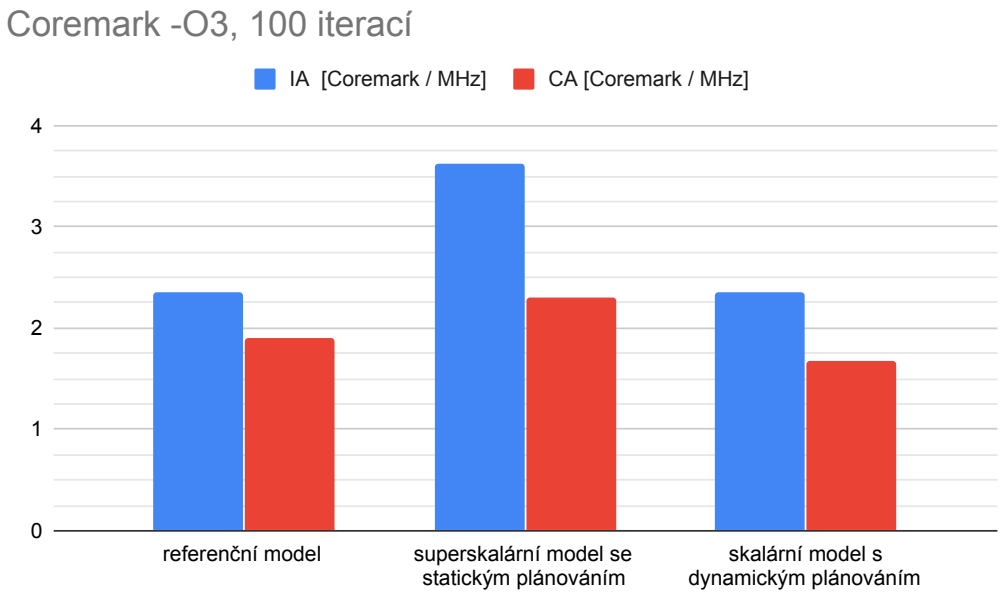
6.2 Test Coremark

Prvním testem, kterým byl ověřován výkon jednotlivých modelů, je test Coremark. Výsledkem tohoto testu je skóre, které zohledňuje frekvenci procesoru, aby bylo možné vyhodnotit efektivitu architektury a nikoliv hrubý výpočetní výkon. Skóre je vyjádřeno v jednotkách *Coremark/MHz*. Při překladu tohoto programu byl zvolen stupeň optimalizace úrovně tři a nebyly využity žádné dodatečné parametry optimalizace. Délka testu byla zvolena na 100 iterací.

Tab. 6.2: Výsledné skóre jednotlivých modelů v testu Coremark

Testovaný model	referenční model	1. navržený model	2. navržený model
Dosažené skóre modelu IA [Coremark/MHz]	2,35	3,62	2,35
Dosažené skóre modelu CA [Coremark/MHz]	1,91	2,31	1,67

V tabulce 6.2 jsou zobrazeny výsledky pro referenční model procesoru Codasip µRISC a pro oba modely navržené v průběhu této práce - první navržený model je procesor se superskalární linkou a statickým plánováním, druhý model se skalární linkou a dynamickým plánováním. Z tabulky je patrné, že první model, který ideálně dosahuje dvojnásobného výkonu, se v praxi blíží skóre ideální skalární zřetězené linky. Druhý model dosahuje i přes opačná očekávání nižšího skóre, než model referenční. Výsledky jsou zobrazeny i v grafu na obrázku 6.2.



Obr. 6.2: Skóre dosažené jednotlivými modely v testu Coremark

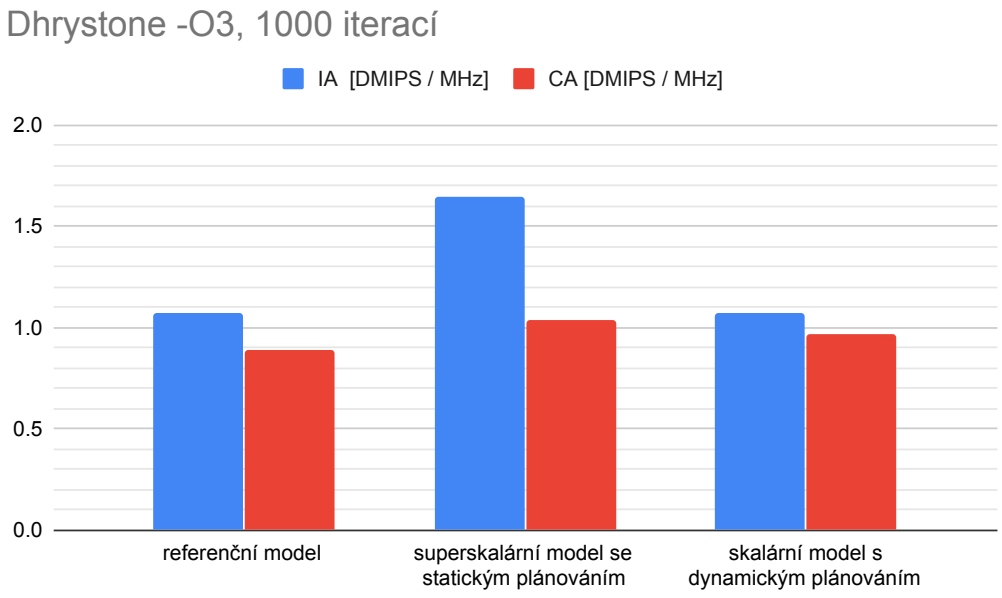
6.3 Test Dhrystone

Druhý výkonnostní test, Dhrystone, je předchůdcem testu Coremark. Pro výsledky tohoto testu platí ještě výraznější závislost na stupni optimalizací při překladu, což bývá označováno jako nevýhodou tohoto testu. Test produkuje několik různých výsledků, pro vyhodnocení efektivity návrhu slouží opět skóre přepočítané podle frekvence procesoru - *DMIPS / MHz*. Test byl taktéž prováděn za využití třetího stupně optimalizace překladu a bez dalších dodatečných parametrů optimalizace. Délka testu byla zvolena na 1000 iterací.

Tab. 6.3: Výsledné skóre jednotlivých modelů v testu Dhrystone

Testovaný model	referenční model	1. navržený model	2. navržený model
Dosažené skóre modelu IA [DMIPS / MHz]	1,07	1,65	1,07
Dosažené skóre modelu CA [DMIPS / MHz]	0,89	1,04	0,97

Tabulka 6.3 zachycuje výsledné skóre jednotlivých modelů, které je zároveň graficky znázorněno na obrázku 6.3. Výsledek prvního modelu je podobný výsledkům testu Coremark, kdy reálný model zřetězené linky vydávající dvě instrukce za takt dosahuje téměř stejného skóre jako ideální model skalární zřetězené linky. Výsledky druhého modelu se však liší a blíží se více teoretickým předpokladům, kdy model vykazuje vyšší skóre v testu a tedy vyšší výkonnost linky, která se více přibližuje ideální lince.



Obr. 6.3: Skóre dosažené jednotlivými modely v testu Coremark

Závěr

Cílem této diplomové práce bylo seznámení se s architekturou procesorů a s jejich instrukční sadou, zejména s procesory typu RISC, seznámení se s principy paralelního vykonávání instrukcí založeném na zřetězeném vykonávání instrukcí, diskutování limitací tohoto přístupu a technik používaných pro zlepšení vlastností reálné zřetězené linky procesoru. Vybrané implementační techniky poté byly demonstrovány na modelu procesoru s redukovanou instrukční sadou Codasip μRISC.

Využitými prostředky při vytváření modelu procesoru byly sada nástrojů Codasip Studio a jazyk pro popis architektur CodAL. Pomocí těchto prostředků byly vytvořeny dva modely založené na referenčním modelu procesoru Codasip μRISC. Pomocí vývojového prostředí Codasip Studio byly vytvořeny model na úrovni instrukční sady i model na úrovni cyklů a z popisu v jazyce CodAL byly vygenerovány všechny nástroje potřebné pro vývoj softwaru i hardwaru. Nakonec byl z modelu na úrovni cyklů vytvořen popis na RTL úrovni v jazyce Verilog.

Zvolenými a implementovanými technikami jsou metoda vydávání více instrukcí za jeden takt a pokročilý algoritmus pro dynamické plánování instrukcí zvaný Tomasulův algoritmus, jež umožňuje vykonávání instrukcí mimo programové pořadí. Obě tyto techniky byly implementovány samostatně pro vyhodnocení přínosu těchto metod nezávisle na sobě. V závěru byly modely podrobeny syntetickým výkonnostním testům Coremark a Dhrystone.

Modely, které byly vytvořeny pro výzkumné účely této práce, nebyly plně verifikovány a je tedy nutné brát v potaz výskyt možných chyb. Nicméně výkonnostní testy použité pro vyhodnocení implementovaných metod obsahují ověření výsledků vypočítaných procesorem při vykonávání algoritmu. Výsledné skóre testů je tedy kontrolováno a lze jej považovat za validní.

Z výsledků referenčního modelu můžeme vypozorovat nemalý rozdíl ve výkonu skutečné zřetězené linky oproti ideální. Úbytek výkonu reálné zřetězené linky je způsoben pokutami za vznik některého z hazardů v lince. První implementovaná technika, metoda vydávání více instrukcí za takt, slouží pro navýšení výkonu ideální zřetězené linky a tudíž i k navýšení výkonu reálné linky. Tento teoretický předpoklad byl výsledky testů potvrzen. Druhá technika, jíž je Tomasulův algoritmus pro dynamické plánování instrukcí, naopak necílí na nárůst výkonu ideální linky, ale jejím hlavním záměrem je minimalizace vzniku hazardů v lince a tudíž maximalizace efektivity reálné linky a přiblížení se výkonu ideální zřetězené linky. Tento teoretický předpoklad se potvrdil v druhém výkonnostním testu, výsledky prvního testu však očekávání nenaplnily. Důvodem může být neoptimální návrh architektury nebo nepoužití dostatečné formy optimalizace překladu programu Coremark. Dalším faktorem je předpoklad, že výhody Tomasulova algoritmu se plně projevují

v architekturách s rozdílnými latencemi instrukcí, přičemž všechny instrukce v procesoru Codasip μRISC mají latenci pouze jeden takt, s výjimkou operací pro práci s pamětí, které mají latenci dva takty způsobenou komunikačním protokolem datové sběrnice.

Další výzkum navazující na tuto diplomovou práci bude pokračovat verifikací vytvořených modelů a opravením případných chyb. Následovat bude proces identifikace faktorů omezujících výkon procesoru a optimalizace navržené architektury. Pro další zkoumání přínosu Tomasulova algoritmu je žádoucí doplnit instrukční sadu procesoru o instrukce s větší latencí a rozšířit počet záznamů v jednotlivých rezervačních stanicích, aby bylo možné naplno zkoumat přínos této metody.

Literatura

[1] P. OPLIVKA. Procesory CISC a RISC: Studijní materiál pro předmět architektury počítačů, Katedra informatiky FEI VŠB-TU Ostrava, 2010. [cit 31.5.2020]. URL: `http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf`.

[2] Modified harvard architecture: Clarifying confusion. [online], September 21, 2015 [cit 31.5.2020]. URL: `http://ithare.com/modified-harvard-architecture-clarifying-confusion/`.

[3] Liu Dake. *Embedded DSP processor design : application specific instruction set processors*. The Morgan Kaufmann series in systems on silicon. Elsevier ; Morgan Kaufmann, Amsterdam : Boston, 2008.

[4] Heath S. *Embedded systems design*. Prentice-Hall, New York, 1. edition, 2003.

[5] John L. a David A. PATTERSON HENNESSY. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Cambridge, MA, 6th edition, 2017.

[6] David A. a John L. HENNESSY PATTERSON. *Computer organization and design: the hardware/software interface*. Elsevier, Cambridge, MA, risc - v edition, 2018.

[7] Nikil DUTT a Prabhat MISHRA. *Processor description languages: applications and methodologies*. Morgan Kaufmann ; Elsevier, Boston: Amsterdam, xxviii edition, 2008.

[8] Codasip s.r.o. *Codasip CodAL Language Reference Manual*, March 2020. Document Version 1.2 (Studio Version 8.3.0).

[9] Codasip s.r.o. *Codasip Studio Technical Reference Manual*, March 2020. Document Version 1.1 (Studio Version 8.3.0).

[10] Codasip s.r.o. *Codasip Studio User Guide*, March 2020. Document Version 2.0 (Studio Version 8.3.0).

[11] Codasip s.r.o. *Codasip uRISC Instruction Set Reference Manual*, March 2020. Document Version 1.0 (Core Version 5.0.0).

[12] Codasip s.r.o. *Codasip Instruction Accurate Model Tutorial*, March 2020. Document Version 1.0 (Studio Version 8.3.0).

[13] ARM. *AMBA 3 AHB-Lite Protocol Specification*, 2006. v1.0. URL: `https://developer.arm.com/docs/ihi0033/a`.

[14] Codasip s.r.o. *Codasip Cycle Accurate Model Tutorial*, March 2020. Document Version 1.2 (Studio Version 8.3.0).

Seznam symbolů, veličin a zkratek

ADL	Jazyk pro popis architektur – Architecture description language
ALU	Aritmeticko logická jednotka – Arithmetic logic unit
CA	Model na úrovni cyklů – Cycle Accurate
CISC	Procesor s komplexní instrukční sadou – Complex Instruction Set Computer
CPI	Počet taktů potřebných k vykonání jedné instrukce – Cycles per Instruction
CPU	Centrální procesorová jednotka – Central processing unit
FPU	Matematický koprocesor – Floating-point unit
HDK	Sada nástrojů pro vývoj hardware – Hardware development kit
HDL	Jazyk pro popis hardwaru – Hardware description language
IA	Model na úrovni instrukční sady – Instruction Accurate
IPC	Počet instrukcí vykonaných za jeden takt – Instructions per Cycle
ISA	Instrukční sada – Instruction set architecture
MISC	Procesor s minimální instrukční sadou – Minimal Instruction Set Computer
RAW	Hazard typu "čtení následující zápis"– Read after write
RISC	Procesor s redukovanou instrukční sadou – Reduced Instruction Set Computer
ROM	Paměť určená pouze ke čtení – Read only Memory
SDK	Sada nástrojů pro vývoj software – Software development kit
UVM	Univerzální verifikační metodologie – Universal Verification Methodology
VLIW	Procesor s velmi dlouhým instrukčním slovem – Very long instruction word
WAR	Hazard typu "zápis následující čtení"– Write after read
WAW	Hazard typu "zápis následující zápis"– Write after write

A Obsah přiloženého souboru .zip

Přiložený soubor formátu .zip obsahuje popis obou modelů navržených při vypracování této diplomové práce na úrovni RTL v jazyce Verilog. Složka obsahující název *model 1* obsahuje model procesoru se statickým plánováním instrukcí popsaném v podkapitole 5.1. V druhé složce, jejíž název je *model 2*, se nachází model procesoru s dynamickým plánováním instrukcí popsaný v podkapitole 5.2.

Obě složky obsahují soubory jazyka Verilog s příponou .v, přičemž nejvyšší modul se nachází v souboru s názvem *codasip_urisc_ca.v*.

```
/ ..... kořenový adresář přiloženého souboru .zip
├── model 1 - RTL (Verilog) ..... adresář se soubory .v
│   ├── codasip_urisc_ca.v ..... nejvyšší modul
│   └── ...
└── model 2 - RTL (Verilog) ..... adresář se soubory .v
    ├── codasip_urisc_ca.v ..... nejvyšší modul
    └── ...
```