FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Implementace superskalární mikroarchitektury ve HDL
Student:	Aleksei Egorov
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

1. Seznamte se s principy činnosti superskalárních mikroarchitektur.
2. Popište činnost Tomasulova algoritmu.
3. Navrhněte superskalární mikroarchitekturu podporující instrukční sadu RV32I (případně její podmnožinu) a implementující Tomasulův algoritmus. Dbejte na parametrizovatelnost návrhu (počet skrytých registrů, velikost reorder buffru apod.).
4. Navrženou mikroarchitekturu popište v HDL (Verilog/SystemVerilog).
5. Simulačně demonstrujte funkčnost Vašeho návrhu.

Jednotlivé kroky konzultujte s vedoucím práce.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 5. prosince 2023 v Praze.

Bakalářská práce

IMPLEMENTACE SUPERSKALÁRNÍ MIKROARCHITEKTURY VE HDL

Aleksei Egorov

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Michal Štepanovský, Ph.D.
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především svému vedoucímu práce Ing. Michalu Štepanovskému, Ph.D. za spoustu konzultací k bakalářské práci, za podporu a smysluplnou zpětnou vazbu, a to i přes hloupost některých mých chyb.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce se věnuje studiu principů fungování superskalárních mikroarchitektur procesorů a návrhu vlastní mikroarchitektury založené na ISA RISC-V RV32I, konkrétně jejímu popisu v jazyce HDL. Vytvořený procesor je v ideálním případě schopen číst z paměti 2 instrukce najednou a dokončit také 2 instrukce najednou. Zdrojové kódy napsané v jazyce Verilog lze použít jako podklad pro budoucí bakalářské práce nebo využít při výuce.

Klíčová slova Tomasulův algoritmus, superskalární procesor, mikroarchitektura, Verilog, instrukční zřetězení

Abstract

This bachelor's thesis is devoted to studying the principles of superscalar processor architectures and designing my own architecture based on ISA RISC-V RV32I, describing it in HDL. The designed CPU is ideally able to read from memory 2 instructions at once and finish also 2 instructions at once. The source codes written in Verilog can be used as a basis for future undergraduate theses or used in teaching.

Keywords Tomasulo algorithm, superscalar processor, microarchitecture, Verilog, instruction chaining

Seznam zkratek

CPU	Central Processing Unit
ALU	Arithmetic-Logic Unit
ISA	Instruction Set Architecture
ROB	Reorder Buffer
RS	Rezervační stanice
FIFO	First In, First Out
HDL	Hardware Description Language
PC	Program Counter
RRSet	Renamed Register Set
ARSet	Architectural Register Set
RRN	Renamed Register Number
ARN	Architectural Register Number

Úvod

S rozvojem a zvyšováním efektivity a výpočetního výkonu počítačových systémů nutně vzniká idea paralelního provádění instrukcí. A nemluvíme pouze o zvyšování počtu procesorových jader, ale také o provádění několika instrukčních vláken v jednom a tomtéž jádře. Všechny moderní procesory tento princip realizují pomocí mnoha technologií, jako je například multithreading, hyperthreading atd.

Hlavní myšlenkou je maximalizovat využití všech výpočetních částí mikroarchitektury procesoru a soustředit se na provádění instrukcí, jejichž operandy má procesor již k dispozici, a ne marně čekat na dokončení instrukcí, jejichž výsledky v danou chvíli nepotřebujeme.

Práce zkoumá a do značné míry rozšiřuje materiály kurzu BI-APS, který jsem měl jako součást svého studijního plánu ve specializaci „Informační bezpečnost“. V tomto kurzu jsem již v rámci semestrální práce v prvním semestru implementoval požadovanou mikroarchitekturu jednocyklového neparalelního procesoru. Bral jsem přednášky tohoto předmětu jako motivaci. Moje práce umožňuje demonstrovat, jak teoretické koncepty a komplikované algoritmy zahrnující složité logické struktury — jako je fronta, výběr nejstaršího prvku z nějaké množiny apod. — lze skutečně realizovat v praxi pomocí jazyka pro popis hardwaru Verilog.

Mým cílem je podrobně prostudovat a analyzovat principy mikroarchitektur superskalárních procesorů a podrobněji Tomasulova algoritmu, navrhnout vlastní model mikroarchitektury a následně jej implementovat. V teoretické části si nejprve projdeme teoretická východiska implementace mikroarchitektur, abychom se čtenáře seznámili s problematikou, základními pojmy a principy fungování. Poté v praktické části popíšeme samotný proces implementace se zaměřením na kritická místa, jejichž obtíže nebyly na první pohled zřejmé.

Výsledky práce lze využít i jako výukový materiál pro předměty, budoucí výzkum nebo navazující práci.

Kapitola 1

Superskalární procesory a Tomasulův algoritmus

Kapitola popisuje základní principy provádění instrukcí v superskalárních architekturách a seznamuje čtenáře s hlavními problémy, s nimiž se návrháři při jejich návrhu setkávají.

Existuje mnoho způsobů, jak zvýšit efektivitu procesoru. Můžeme například optimalizovat kompilaci programového kódu do instrukčního kódu dané ISA. Ve zřetězených procesorech při zahájení x + 1 kroku zpracování instrukce začne datová cesta zároveň vykonávat x krok zpracování následující instrukce a využívá se tím **paralelismus v čase**. [1] A nebo můžeme zmultiplikovat některé kritické části mikroarchitektury a snažit se dosáhnout jejich nejefektivnějšího využití a tím využijeme **paralelismus v prostoru**. Posledně jmenovaný přístup se používá při vývoji superskalárních procesorů. Právě na ně se zaměříme v této práci nejvíce.

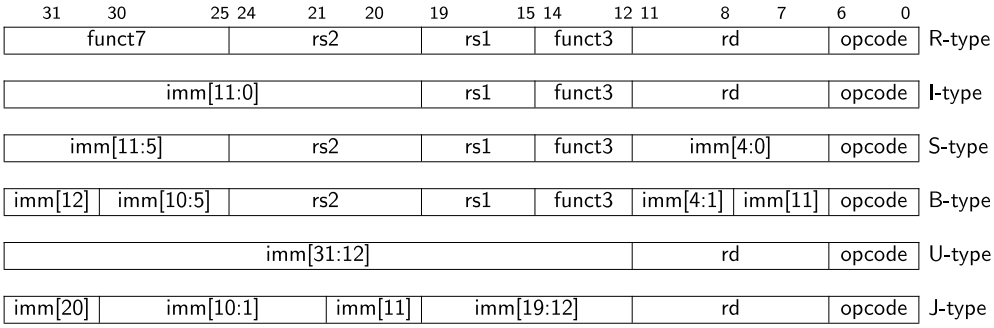
1.1 Kódovaní instrukcí v ISA RISC-V RV32I

▶ **Definice 1.1.** *Instruction Set Architecture je abstraktní rozhraní mezi hardware a nízkoúrovňovým software, které zahrnuje veškeré informace nezbytné k psaní korektních programů ve strojovém jazyce (nebo jazyce symbolických adres). [2]*

Mikroarchitektura, kterou jsem navrhl, vychází z ISA RISC-V základní specifikace RV32I. Jedná se o ISA vyvinuté na základě výzkumu Kalifornské univerzity v Berkeley. Základní specifikace RV32I obsahuje sadu 32 registrů o šířce 32 bitů a 39 instrukcí, které jsou kódovány v 6 různých formátech. Podrobnosti viz obrázek 1.1.

Různá kódování jsou způsobena tím, že instrukce berou data pro své operandy z různých zdrojů. Za rozpoznání je obvykle zodpovědný Instruction

dekoder. Podívejme se blíže na jednotlivé formáty.



Obrázek 1.1 Kódovaní instrukcí v instrukční sadě RISC-V RV32I. [3]

R-type aritmetické a logické instrukce používající jako zdroj operandů pouze registry.

I-type aritmetické a logické instrukce. Jeden z operandu se bere přímo z instrukce jako rozšířené 32-bitové znaménkové číslo. Druhý operand je z registru.

S-type store-instrukce, ukládající hodnotu registru do paměti.

B-type podmíněné skokové instrukce. Jeden z operandu se bere přímo z instrukce jako rozšířené 32-bitové znaménkové číslo. Druhý operand je z registru.

U-type aritmetické a logické instrukce, které berou jenom přímý operand.

J-type nepodmíněné skokové instrukce, které berou jenom přímý operand.

Dále se zaměříme pouze na celočíselné aritmetické instrukce typu registr-registr (formát R-type), neboť ve své implementaci mám pouze je. Na obrázku 1.2 je vidět kódování R-type instrukcí s příklady. *Opcode* všech těchto instrukcí je stejný, typ operace je určen hodnotami *funct3* a *funct7*.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Obrázek 1.2 Kódování R-type instrukcí. [3]

ADD a SUB provádějí sčítání, resp. odčítání. Přetečení jsou ignorována a hodnota XLEN bitů výsledků jsou zapsány do cíle. SLT a SLTU provádějí

znaménkové, resp. bez znaménkové operace a zapisují 1 do *rd*, pokud *rs1* < *rs2*, a 0 v opačném případě. AND, OR, a XOR provádějí bitové logické operace. SLL, SRL a SRA provádějí logický posun doleva, logický posun doprava a aritmetický posun doprava na hodnotě v poli registru *rs1* o hodnotu posunu uloženou v dolních 5 bitech registru *rs2*. [3]

1.2 Definice a principy fungování superskalárních CPU

▶ **Definice 1.2.** *Central Processing Unit, také centrální procesor nebo jen procesor, je elektronický obvod, který vykonává instrukce počítačového programu, jako jsou aritmetické, logické, řídicí a vstupně-výstupní (I/O) operace. [4]*

1.2.1 Logické stupně provádění instrukcí v procesoru

Podívejme se na hlavní fáze provádění instrukcí v procesoru.

Instruction fetch (IF). Procesor načítá jednu nebo více instrukcí z paměti. Adresa je uložena v program counteru.

Instruction decode (ID). Procesor dekóduje obsah instrukce. Určí její typ, registry, se kterými bude pracovat, přítomnost a obsah přímých operandů.

Execution (EX). Samotné provádění instrukcí. Provádění aritmetických a logických operací v ALU, přenastavení program counteru.

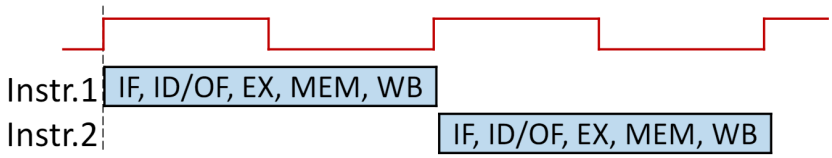
Memory access (MEM). V případě instrukcí pracujících s pamětí, procesor přebírá z EX fáze před vypočítanou efektivní adresu a ukládá/stahuje hodnotu z dané adresy v paměti počítače.

Write back (WB). V případě zapsaní dat do registru, procesor tam zapisuje výsledek z EX fáze. [5]

1.2.2 Typy procesoru z pohledu paralelního provádění instrukcí

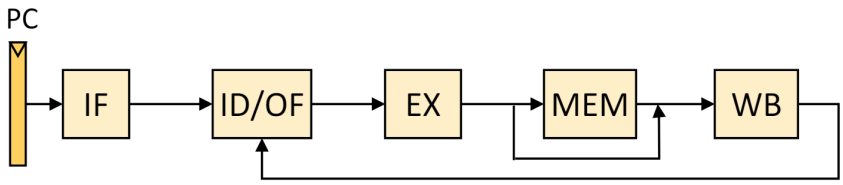
1.2.2.1 Jednocyklový procesor

V jednocyklovém procesoru se všechny fáze provedou během jednoho taktu (obrázek 1.3). Nová instrukce se začne zpracovávat až po dokončení staré. Na schématu 1.4 je vidět, že jednocyklový procesor nemá mezistupňové registry. Z tohoto důvodu nemusí návrhář řešit žádné problémy se synchronizací EX, MEM a WB fází. Provádění instrukcí není nikdy pozastaveno.



Obrázek 1.3 Souvislost provádění jednotlivých stupňů a taktů v jednocyklovém procesoru. [6]

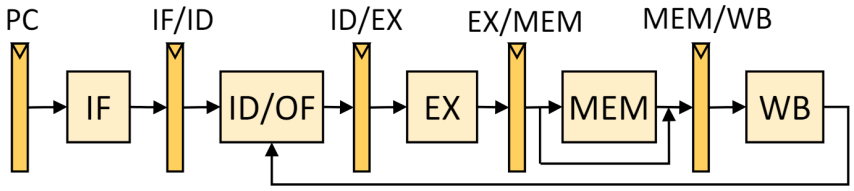
Z hlediska počtu provedených instrukcí za takt jednocyklové procesory jsou v průměru nejhorší. Zatímco jedna část provádí operace, všechny ostatní části jsou nečinné a čekají na další instrukce.



Obrázek 1.4 Schéma provádění instrukcí v jednocyklovém procesoru. [6]

1.2.2.2 Zřetězený procesor

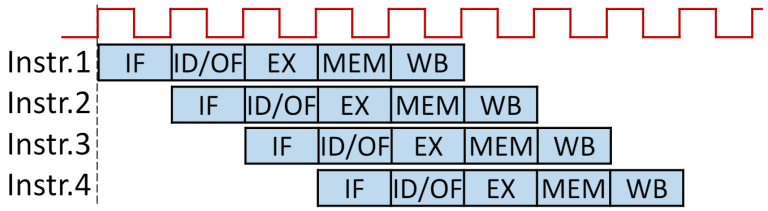
Ve zřetězeném procesoru jednotlivé fáze jsou rozdělené do několika stupňů pomoci mezistupňových registrů, které si pamatují výsledek provedení předchozí fáze. To umožňuje načítání a dekódování dalších instrukcí z paměti během výpočtu ještě nedokončené instrukce. V takové mikroarchitektuře už musí návrhář zajistit detekci různých typů synchronizačních hazardů. V určitých případech a konfiguracích se provádění instrukcí pozastavuje, dokud neproběhnou všechny potřebné změny hodnot ve stupních MEM a WB.



Obrázek 1.5 Schéma provádění instrukcí ve zřetězeném procesoru. [6]

Z hlediska počtu provedených instrukcí za takt je zřetězený procesor v průměru mnohonásobně efektivnější než jednocyklový, protože často umožňuje

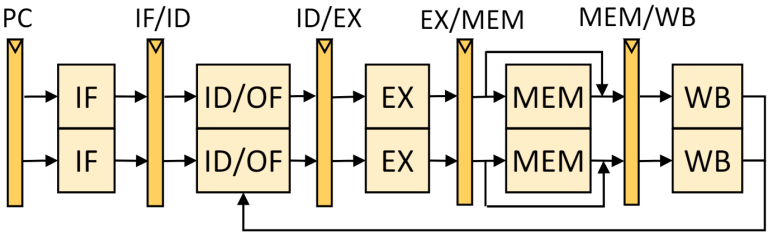
využít jeden takt procesoru k provádění operací ve všech stupních současně. Části procesoru jsou nečinné pouze v případě detekce hazard.



Obrázek 1.6 Souvislost provádění jednotlivých stupňů a taktů ve zřetězeném procesoru. [6]

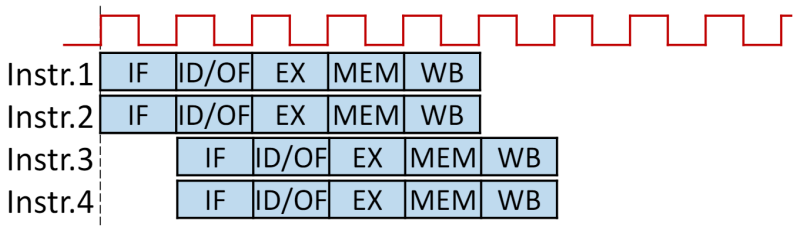
1.2.2.3 Superskalární procesor

Superskalární procesor je do jisté míry rozšířením koncepce zřetězeného procesoru. Mezi komponentami provádějícími různé logické stupně jsou registry, které si pamatují výsledky, ale některé komponenty mají více instancí, které umožňují zpracování více instrukcí v jednom taktu. Stejně jako ve zřetězeném procesoru je třeba detekovat různé typy hazardů a řídit instrukční tok.



Obrázek 1.7 Schéma provádění instrukcí ve zřetězeném procesoru. [6]

Z hlediska počtu provedených instrukcí za takt je superskalární procesor v průměru nejlepším z uvažovaných typů. Dále budeme hovořit o různých typech architektur superskalárních procesorů. Některé typy využívají Tomasulův algoritmus, který umožňuje využít některé komponenty i v případě detekce hazard.



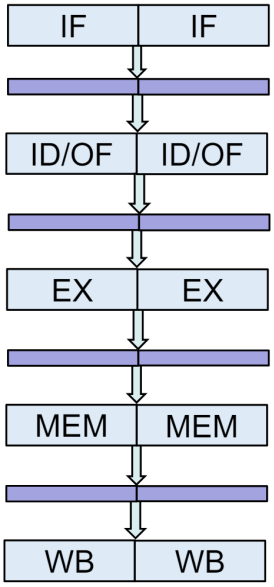
Obrázek 1.8 Souvislost provádění jednotlivých stupňů a taktů ve zřetězeném procesoru. [6]

1.2.3 Superskalární procesor

▶ **Definice 1.3.** *Superskalární procesor je specifický typ mikroprocesorů, který využívá paralelismus na úrovni instrukcí, což usnadňuje provádění více než jedné instrukce během jednoho taktu a má při tom víc než jednu frontu pro zřetězené zpracování.*

▶ **Definice 1.4.** *Šířka zřetězení je počet instrukcí, které mohou být přineseny, dekódovány nebo dokončeny v každém cyklu. [6]*

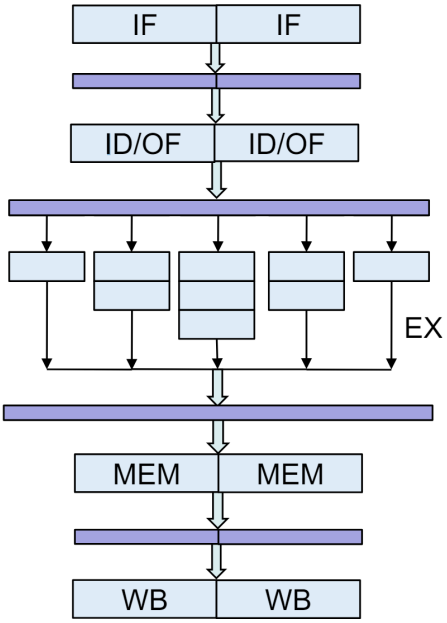
Rozlišujeme mezi **unifikovaným** a **diverzifikovaným** superskalárním zřetězením. Při unifikovaném zřetězení mají souběžně prováděné instrukce stupně EX prováděnou samostatnými nezávislými komponentami. Příklad je demonstrován na obrázku 1.9.



Obrázek 1.9 Unifikované superskalární zřetězení šíře 2. [6]

Při diverzifikovaném zřetězení je stupeň EX diverzifikovaná a jednotlivé výpočetní části stupně EX mohou být určeny k provádění různorodých operací. Například první samostatná ALU může je odpovědná za aritmetické operace a druhá samostatná ALU počítá adresy pro paměť. Vzor je demonstrován na obrázku 1.10.

Téměř všechny moderní superskalární architektury jsou **dynamické**. Na rozdíl od **statických** umožňují provádění instrukcí „out-of-order“, což znamená, že se neprovádí následující instrukce v programu, ale další instrukce, která je připravena k provedení. Právě tato mikroarchitektura je realizovaná v mojí implementaci. Tento princip umožňuje minimalizovat pozastavení instrukčního toku a eliminovat negativní vliv některých synchronizačních ha-



Obrázek 1.10 Diverzifikované superskalární zřetězení šíře 2. [6]

zardů. Podrobněji se tím budeme zabývat v části věnované Tomasulovu algoritmu.

1.2.4 Logické stupně superskalárního instrukčního zřetězení

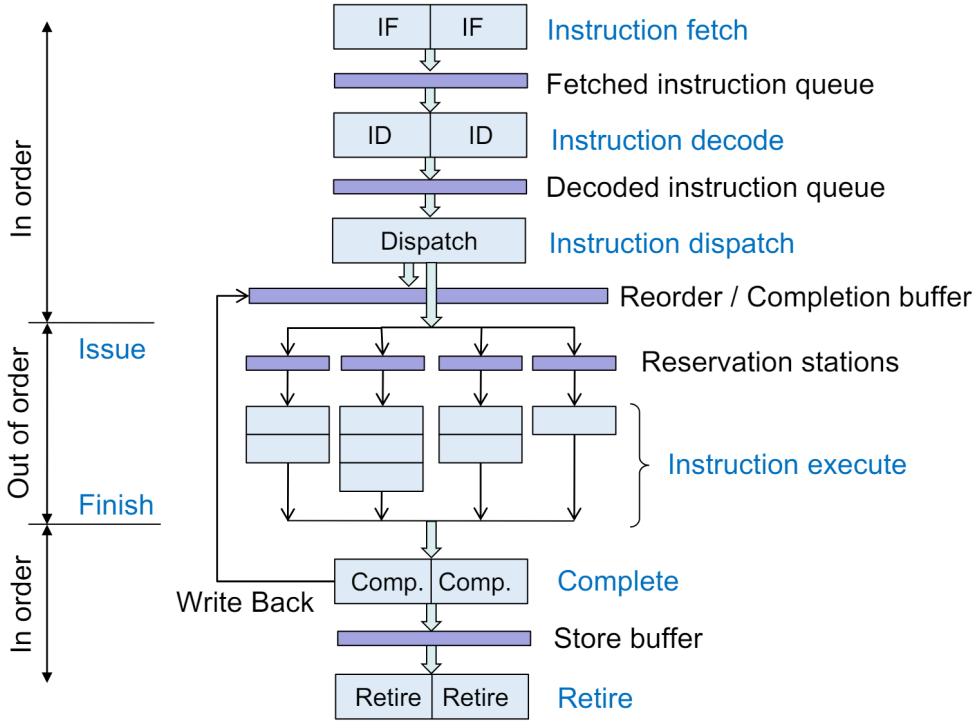
Před podrobnějším zkoumáním Tomasulova algoritmu se podíváme, v čem se logické stupně superskalárního instrukčního zřetězení liší od stupňů konvenčního skalárního zřetězení. Existuje 6 fází, které však nemusí nutně odpovídat skutečnému fyzickému rozmístění komponent v architektuře. [6] Na obrázku 1.11 se můžeme podívat jak ony přibližně spolu souvisí.

1.2.4.1 Instruction fetch

Ve fázi načtení instrukcí (*Instruction fetch*) z paměti se v každém taktu načítá počet instrukcí rovný šířce zřetězení. V této fázi mohou v různých ISA nastat různé problémy. Například při různé délce instrukcí je třeba počítat s paddingem. [6] Navíc, čím větší je šířka zřetězení, tím dražší bude hardware a tím větší plochu čipu budou zabírat spoje mezi komponentami a buffery.

1.2.4.2 Instruction decode

Ve fázi dekódování instrukcí (*Instruction decode*) je třeba identifikovat čtené instrukce. Procesor musí určit jejich typ, jaké operace provádějí, jaké operandy



Obrázek 1.11 Logické stupně superskalárního instrukčního zřetězení. [6]

instrukce obsahuje a kde v bajtech se ony nachází. Jedním z hlavních úkolů této fáze je odhalit meziinstrukční závislosti. O tom, jak procesor reaguje na detekci takových závislostí, bude podrobněji pojednáno ve kapitole 2.

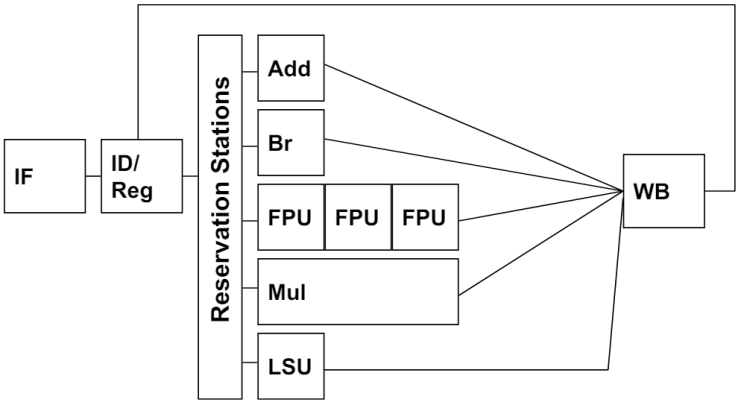
1.2.4.3 Instruction dispatch

▶ **Definice 1.5.** *Rozřazení je proces přejmenování registrů a alokace prostředků v ROB a v rezervační stanici (RS). [6]*

Počínaje fází rozřazování instrukcí (*Instruction dispatch*) se dostáváme do zóny „out-of-order“. Na základě toho, jak byly instrukce dekódovány, jsou dále distribuovány do RS. O způsobu uspořádání rozhoduje návrhář v závislosti na technickém zadání, účelnosti.

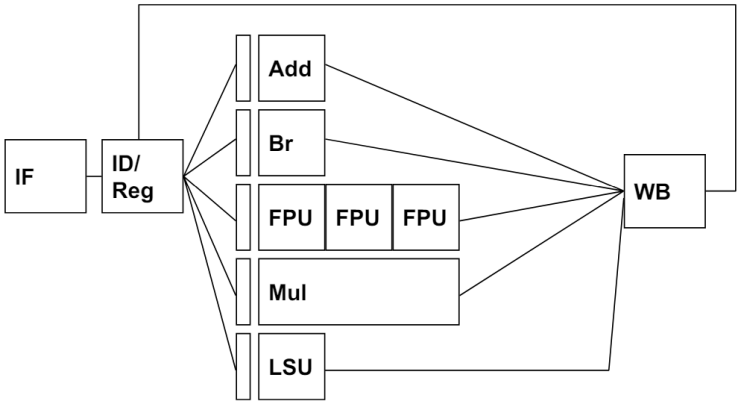
Centralizované/sdílené rezervační stanice umožňuje všem instrukcím sdílet jednu velkou rezervační stanici obvykle stejnou velikostí jako ROB, a dosáhnout nejlepšího celkového využití její kapacity. Schéma mikroarchitektury s centralizovanou rezervační stanicí, která byla použitá v procesoru Intel Pentium Pro, je znázorněno na obrázku 1.12. Ale jako zápor vyžaduje centralizované řízení a složitější hardware. Náročnost na hardware se obvykle snižuje tím, že několik ALU stejné specializace se skupuje pod jeden

port vycházející z RS. [6]



Obrázek 1.12 Centralizované rezervační stanice (Intel Pentium Pro) [7]

Distribuované/dedikované předpokládá, že rezervačních stanic je tolik, kolik je specializací ALU a každá rezervační stanice má pouze jeden výstupní port. Schéma mikroarchitektury s distribuovanými rezervačními stanicemi, které byly použité v procesoru PowerPC 604, je znázorněno na obrázku 1.13. Potenciálně horší využití celkové vykonávací kapacity, pokud některé rezervační stanice v některých větvích budou saturovány, zatímco v jiných větvích budou volné. [6]

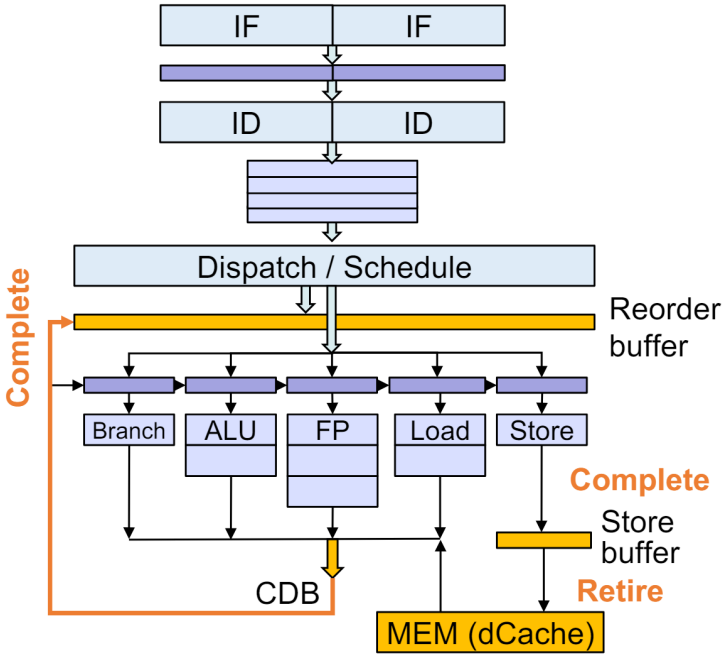


Obrázek 1.13 Distribuované/dedikované rezervační stanice (PowerPC 604) [7]

Hybridní způsob organizace je kompromisní mezi centralizovaným a distribuovaným. Rezervačních stanic může být několik a každá může mít několik výstupních portů.

1.2.4.4 Instruction execute

Ve fázi vykonání instrukcí (*Instruction execute*) hrají hlavní roli 2 typy komponent: rezervační stanice a vykonávací jednotky (ALU). Nejprve instrukce v rezervačních stanicích čekají, až budou všechny jejich operandy připraveny k dalším operacím. Poté je instrukce odeslána do pro ni specializované ALU, kde se obvykle nachází několik cyklů. V superskalárních procesorech jsou ALU obvykle specializovány podle typu prováděných operací. Například na obrázku 1.14 jsou ALU zodpovědné za skokové, aritmetické, float a pamětové operace.



Obrázek 1.14 Podrobnější schéma specifikace RS a fází *Complete* a *Retire*. [6]

1.2.4.5 Instruction complete

Ve fázi dokončení instrukcí (*Instruction complete*) instrukce je považována za dokončenou, když ukončí vykonávání (opustí EX stupeň) a aktualizuje architekturální stav procesoru. [6]

1.2.4.6 Instruction retire

Fáze odbavení instrukcí (*Instruction retire*) je určená výhradně pro instrukce, které zapisují data do paměti. V mé mikroarchitektuře se jí nedotkneme. Fáze odbavení instrukcí končí odesláním dat určených k zápisu do zápisové fronty.

1.3 Tomasulův algoritmus

Konečně se dostáváme k Tomasulovu algoritmu a jeho roli při vykonaní instrukcí mimo programové pořadí. Rozlišujeme různé závislosti, které brání tomu, aby instrukce přešla do dalšího stupně provádění. V zřetězené mikroarchitektuře vede zastavení instrukce v jednom stupni i k zastavení všech předchozích stupňů. Hlavním cílem algoritmu je eliminovat některé závislosti mezi instrukcemi tak, aby se celé zřetězení pozastavovalo co nejméně.

1.3.1 Závislosti mezi instrukcemi

Existují 3 typy závislostí, které mohou negativně ovlivnit propustnost mikroarchitektury.

1.3.1.1 Datové závislosti

Datová závislost nastává, když instrukce odkazuje na data některé předchozí instrukce. [6] Právě s takovými závislostmi se Tomasulův algoritmus vypořádává. Přestože se snažíme spouštět co nejvíce instrukcí paralelně, navenek pro programátora by to mělo vypadat, že vše běží „in-order“.

Rozlišujeme 3 typy datových závislostí, které mohou způsobit 3 typy datových hazardů. Ukažme si je na příkladech kódu. Představme si, že všechny instrukce jsou prováděny paralelně a `xN` označujeme práci s obsahem registru, kde `N` je jeho číslo.

WAW (write after write) může nastat, když paralelně prováděné instrukce zapisují data do jednoho registru nebo položky paměti. Představme si, že ve výpisu 1.1 `sub` se provede rychleji než `add`. Pak bude registr obsahovat hodnotu provedení `add`, která by ve skutečnosti měla být přepsána. Lze řešit přejmenováním registrů pomocí Tomasulova algoritmu.

Výpis kódu 1.1 Příklad programu s WAW závislostí.

```
add     x0, x1, x2
mult    x3, x0, x2
sub     x0, x1, x0
```

WAR (write after read) může nastat, když druhá z paralelně prováděných instrukcí zapisuje data do registru, z něhož čte první instrukce. Představme si, že ve výpisu 1.2 `addi` se provede rychleji než `mult`. Pak `mult` špatně přečte hodnotu `x2`. Lze řešit přejmenováním registrů pomocí Tomasulova algoritmu.

RAW (read after write) může nastat, když první z paralelně prováděných instrukcí zapisuje data do registru, z něhož čte druhá instrukce. Představme

Výpis kódu 1.2 Příklad programu s WAR závislostí.

```
mult    x1, x2, x3
addi    x2, x0, -40
```

si, že ve výpisu 1.3 `mult` se začne provádět rychleji než bude dokončena `mv`. Pak `mult` špatně přečte původní hodnotu `x0`, která by ve skutečnosti měla být přepsána v `mv`. Nelze řešit přejmenováním registrů, nelze eliminovat.

Výpis kódu 1.3 Příklad programu s RAW závislostí.

```
mv      x0, x1
mult    x3, x0, x2
```

1.3.1.2 Řídicí závislosti

Řídicí závislosti vznikají, když jsou v kódu programu přítomny skokové instrukce.

Výpis kódu 1.4 Příklad programu s řídicími meziinstrukčními závislostmi.

```
jalr    x0, x1, LABEL
add     x0, x1, x2
```

Představme si, že na ukázce kódu 1.4 `add` se provede, přestože `jalr` měla změnit PC. V tomto případě je hodnota `x0` nesprávná. Řídicí hazardy se řeší spekulativním vykonáváním instrukcí. [6]

1.3.1.3 Strukturální závislosti

Nastane, když počet instrukcí čekajících na volnou výpočetní jednotku přesáhne počet těchto výpočetních jednotek. V takovém případě jsou instrukce nuceny obsadit místo v bufferech před fází EX a zabírají místo, do kterého již mohly být zapsány další instrukce.

Řešení strukturálních hazardů je zcela prozaické. Při návrhu mikroarchitektury se můžeme snažit přidat co nejvíce ALU a spojů. Kvalita řešení je dána nákladností čipů.

1.3.2 Činnost Tomasulova algoritmu

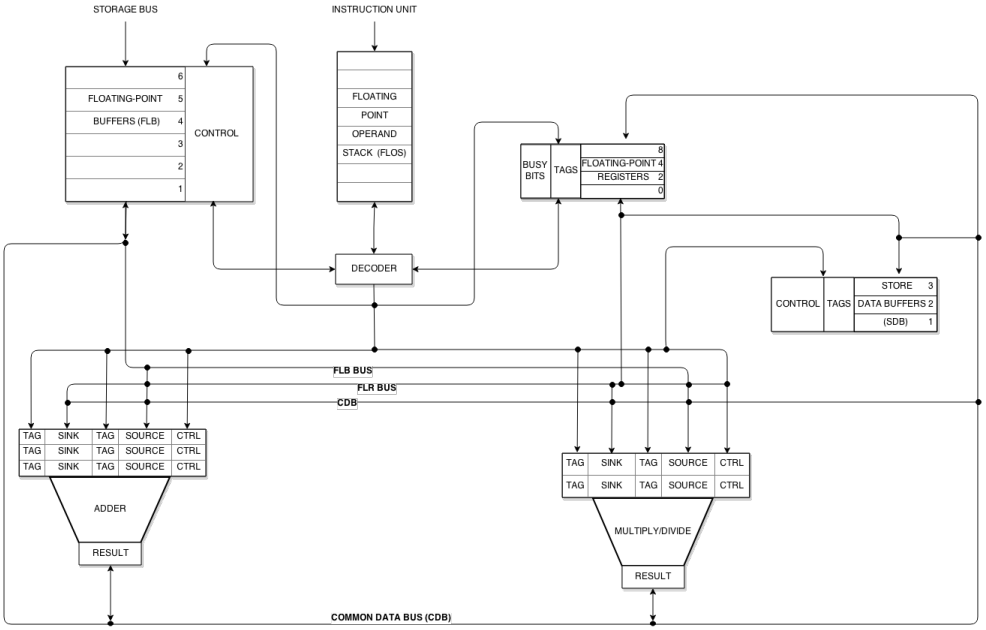
▶ **Definice 1.6.** *Tomasulův algoritmus je hardwarový algoritmus procesorové mikroarchitektury pro dynamické plánování instrukcí, který umožňuje prová-*

Výpis kódu 1.5 Zatížení ALU, zodpovědné za MULT operace.

```
mult      x0, x0, x0
mult      x0, x0, x0
mult      x0, x0, x0
mult      x0, x0, x0
mult      x0, x0, x0
mult      x0, x0, x0
```

dění instrukcí mimo pořadí a umožňuje efektivnější využití více vykonávacích jednotek. [8]

Vyvinul jej Robert Tomasulo ve společnosti IBM v roce 1967 a poprvé byl implementován v jednotce *IBM System/360 Model 91* s plovoucí desetinnou čárkou. [8] Ukázka té implementace je na obrázku 1.15.



Obrázek 1.15 Struktura první implementace Tomasulova algoritmu v *IBM System/360 Model 91's floating point unit* v roce 1967. [8]

Vidíme, že tato implementace specializuje 2 ALU na operace sčítání a násobení/dělení.

1.3.2.1 Skryte registry

Tomasulův algoritmus řeší problém WAR a WAW závislostí přidáním tzv. skrytých registrů do mikroarchitektury. Ty jsou pro programátora neviditelné

a slouží k **přejmenování** architekturálních registrů.

Idea: Nechť nějaká instrukce zapisuje do architekturálního registru *Xi*. Tento registr bude přejmenován na *RRXj*. Všechna následná čtení z *Xi* se nahradí čtením z *RRXj*, dokud nenastane opět zápis do *Xi* (registr *Xi* v takovém případe bude opět přejmenován). Registr *RRXj* se pokaždé volí tak, aby nekolidoval s již používanými registry pro přejmenování. [6]

Pro přejmenování se obvykle používá soubor skrytých registrů, který má stejnou velikost jako soubor architekturálních registrů. Informace o jejich mapování jsou obsaženy v Reorder bufferu, který si pamatuje také stav každé instrukce a jejího cílový registr. Reorder bufferu je obvykle realizován jako FIFO fronta. Po dokončení provádění instrukce je výsledek zapsán do Reorder bufferu a odpovídajícího mapovaného fyzického skrytého registru. Pokud je na počátku Reorder bufferu nedokončená instrukce, blokuje dokončení dalších instrukcí. Když instrukce na počátku Reorder bufferu je dokončena, výsledek instrukce se zapíše do příslušného architekturálního registru a mapování mezi logickým a fyzickým registrem se vymaže. Pokud je Reorder buffer plný, vydání instrukce se zastaví. [5, 9]

1.3.2.2 Základní fáze Tomasulova algoritmu

Tomasulův algoritmus lze rozdělit do 4 logických fází.

Dispatch Instrukce přechází z fronty dekódovaných instrukcí do vydání, pokud existuje volná položka v Reorder buffer (ROB), rezervační stanici (RS) i souboru skrytých registrů (RRSet). Do příslušné položky v rezervační stanici se zatímco načítají výsledky předchozích instrukcí, které se ve stejném cyklu zapisují do Common Data Bus (CDB). Ve všech komponentách, kde se alokovalo místo, nastaví se odpovídající *Busy* bity.

Issue Instrukce shromažďuje všechny hodnoty, které potřebuje, z CDB. Pokud by na konci cyklu měla instrukce všechny operandy k dispozici, přejde do fáze vykonávání. Instrukce však musí zůstat ve fázi *Issue*, pokud existuje starší instrukce (tj. vyslaná dříve) přiřazená ke stejné ALU, která je rovněž připravena přejít do fáze vykonávání.

Execute V této fázi instrukce opouští RS a přechází do ALU. Výpočet výsledku může trvat několik cyklů v závislosti na specializaci a vnitřní organizace ALU.

Writeback Jakmile se výpočet instrukce dokončí, zapíše se její výsledek do CDB. Při zpětném zápisu se v souboru registrů aktualizuje příslušný registr. Během této fáze také jakákoli instrukce v RS, která čeká na tuto hodnotu, aktualizuje své hodnotové pole v RS. [6, 9]

Kapitola 2

Návrh vlastní mikroarchitektury superskalárního procesoru

Tato kapitola popisuje mou vlastní implementaci mikroarchitektury založené na ISA RISC-V RV32I. Seznámíme se s každou komponentou zvlášť a popíšeme detaily jejich propojení.

Moje mikroarchitektura má za účel demonstrovat principy práce Tomasulova algoritmu.

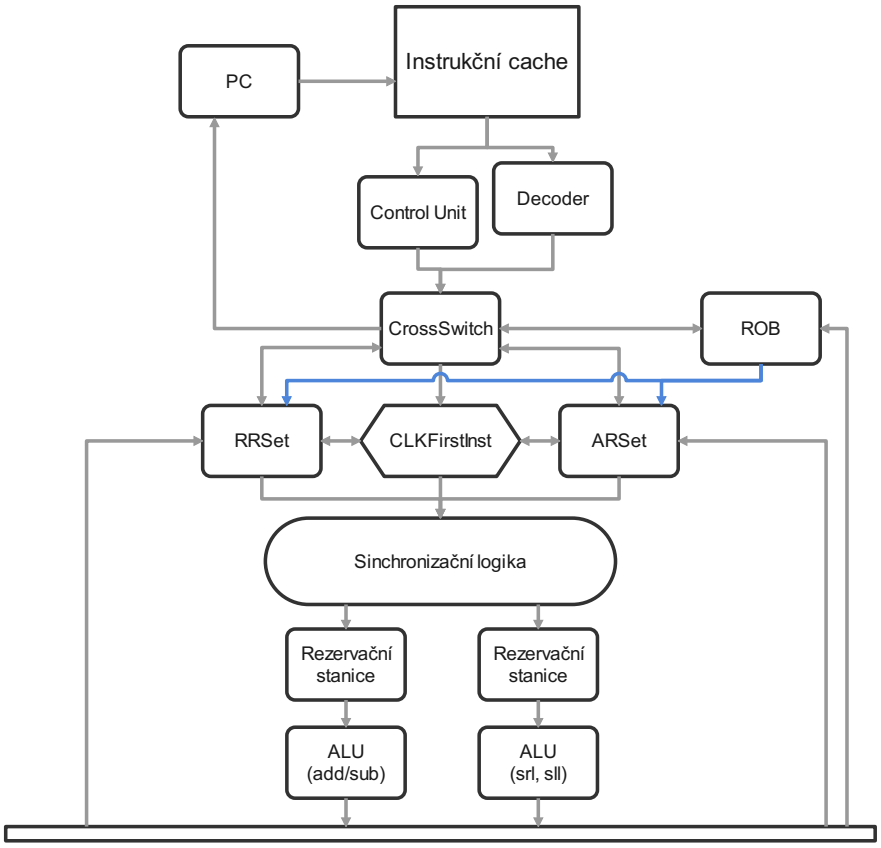
Instrukční zřetězení má šířku 2, tj. procesor může z instrukční paměti číst až 2 instrukce najednou. V jednom taktu lze také vykonávat až 2 instrukce ve výpočetních jednotkách.

Vzhledem k tomu, že mikroarchitektura je určena pouze k demonstraci činnosti Tomasulova algoritmu, má některá zjednodušení vzhledem k procesorům, které implementují celou ISA RISC-V RV32I. Konkrétně: jsou implementovány pouze 4 aritmeticko-logické instrukce jednoho typu R-type, skokové instrukce nejsou nijak řešeny, takže procesor nepodporuje programy, které provádějí nějaké skoky. Pro jejich implementaci by bylo nutné mikroarchitekturu do značné míry modifikovat a přidat komponentu zodpovědnou za označování a spekulativní provádění instrukcí. Vyřešeny nejsou také instrukce, které pracují s pamětí. Paměťové ani skokové instrukce nijak zvlášť neovlivňují činnost Tomasulova algoritmu a implementované instrukce plně umožňují demonstrovat princip „out-of-order“ provádění.

2.1 Procházení celým instrukčním cyklem

Implementoval jsem 32 registrů , program counter a 4 instrukce, což mi umožnilo demonstrovat paralelizaci jejich provádění jako důsledek práce Tomasu-

lova algoritmu. Všechny tyto 4 implementované instrukce jsou kódovány ve formátu celočíselných aritmetických instrukcí typu registr-registr (formát R-type). Navrhuji projít hlavní fáze instrukčního zřetězení a podívat se, jak moje mikroarchitektura implementuje každou z nich.



Obrázek 2.1 Navržena implementace superskalární mikroarchitektury.

2.1.1 Instruction fetch

Na začátku běhu se provede celkový reset způsobem znázorněným na výpisu 2.1. Procesor je schopen číst instrukce z paměti až po prvním taktu.

Výpis kódu 2.1 Reset procesoru při zapnutí.

```
reset <= 1;
#4;
reset <= 0;
```

Instrukce jsou uložené ve formátu *.hex* jako je ukázáno na výpisu 2.2. Procesor je čte 2 řádky za takt a předává je k dekódování.

Výpis kódu 2.2 Příklad programu pro navřenou mikroarchitekturu.

```
00208033        // add x0, x1, x2
002051b3        // srl x3, x0, x2
40008033        // sub x0, x1, x0
003010b3        // sll x1, x0, x3
00000133        // add x2, x0, x0
003010b3        // sll x1, x0, x3
```

2.1.2 Instruction decode

Instruction decoder je komponenta odpovědna za fázi dekódovaní. Decoder vybírá z instrukcí hlavní údaje o zdrojových a cílových registrech a předává je do stupně Instruction dispatch. Představím její interface na výpisu 2.3.

Výpis kódu 2.3 Vnitřní struktura dekodéru.

```
module InstDecoder( input  [31:0]   Instructions    [1:0],
                    output [2:0]    SrcOp1          [1:0],
                    output [2:0]    SrcOp2          [1:0],
                    output [2:0]    SrcTarget       [1:0]);
```

Kde *Instructions* je vstup dvou 32-bitových instrukcí. *SrcOp1* a *SrcOp2* jsou výstupy, které obsahují zdrojové registry. *SrcTarget* je výstup, který obsahuje cílový registr.

2.1.3 Instruction dispatch

V této fázi projdou 2 instrukce přes CrossSwitch, který podle typu příchozí instrukce ji posílá dále na odpovídající port do rezervačních stanic podle typu operací. Cílový registr je přejmenován a v komponentách jsou alokovány všechny potřebné položky (podrobněji v podkapitole o komponentách). Z registrů se načtou potřebná data pro operandy.

V mé práci mají rezervační stanice distribuční organizaci. To znamená, že každá RS obsahuje instrukce pouze stejné specifikace.

2.1.4 Instruction execute

Jakmile je instrukce připravena (má přítomné a validní oba operandy), odešle se do výpočetní jednotky.

2.1.5 Instruction complete

Když výpočetní jednotka dokončí provádění instrukce, výsledky se odešlou do CDB. Všechny příslušné komponenty aktualizují svá data, dealokuji příslušné položky.

2.2 Vyvinuté komponenty procesoru

2.2.1 CrossSwitch

CrossSwitch je křížový přepínač, který přebírá nejen instrukce z dekodéru, ale také údaje o volném místě v komponentách: RRSet, ROB a rezervačních stanic — protože právě CrossSwitch je zodpovědný za pozastavení celého instrukčního zřetězení. Jediná součástka, která má vliv na PC. Pokud alespoň v některé komponentě nejsou 2 volná místa, PC se nezvýší a procesor čeká na konec provádění aktuálních instrukcí.

Výpis kódu 2.4 Přeuspořádání toku instrukcí v křížovém přepínači.

```
module CrossSwitch( input    [2:0]   SrcOp1    [1:0],
                    input    [2:0]   SrcOp2    [1:0],
                    input    [2:0]   SrcTarget [1:0],
                    input    [4:0]   SrcCtrl   [1:0],
                    ...
                    output reg  [13:0]  ToRS1Way0,
                    output reg  [13:0]  ToRS1Way1,
                    output reg  [13:0]  ToRS2Way0,
                    output reg  [13:0]  ToRS2Way1);
```

2.2.2 PC

PC obsahuje adresu instrukce, která bude načtena z paměti v příštím hodinovém cyklu. V mé architektuře se pouze inkrementuje v každém taktu nebo se nemění vůbec v případě nedostatku místa v některých komponentách.

2.2.3 Reorder buffer

Reorder buffer je komponenta odpovědná za koordinaci ostatních komponent ve fázích Instruction dispatch, Instruction execute a Instruction complete. Je implementována jako FIFO kruhová fronta, tj. jedná se o „in-order“ buffer, který se řídí ukazateli Head a Tail. Pokud ROB má alespoň 2 volná místa, pak na začátku každého taktu obdrží jako vstup 2 instrukce a alokuje jim 2 položky.

Právě tato komponenta obsahuje mapování obou souborů registrů, takže zapamatuje si čísla architekturálního cílového i přejmenovaného cílového. Rovněž ROB je zodpovědná za změnu stavů jednotlivých instrukcí. Pokud je instrukce vydána resp. dokončena ALU, je bit Issued resp. Finalised nastaven na 1. Pokud je finalizována instrukce, na kterou ukazuje Head, odpovídající položka v RRSet se dealokuje a položka v ARSet se obnoví na validní. Struktura Reorder buffer je uvedena v tabulce 2.1.

Tabulka 2.1 Stav ROB na při zapnutí procesoru (skutečně má 8 řádků). [6]

B	Iss	Fin	ARN	RRN
0				
0				
0				

Kde ARN — číslo architekturálního registru, RRN — číslo skrytého registru. Head a Tail se na začátku ukazují na první položku.

2.2.4 Rezervační stanice

Rezervační stanice je fronta, která je bufferem pro instrukce čekající na volné místo v odpovídající ALU. Nejčastěji do RS přicházejí instrukce, které nejsou připraveny k dalšímu výpočtu.

RS pro nové instrukce alokuje položky, zapamatuje si, jaký typ operace odpovídá každé instrukci, cílový přejmenovaný registr a platnost každého ze dvou operandů a jejich hodnotu. Pokud je bit platnosti operandu 0, pak pole s jeho hodnotou obsahuje číslo skrytého registru, který tento operand aktualizuje, když přijde z ALU. Rezervační stanice také předává křížovému přepínači informaci o své volné kapacitě.

Tabulka 2.2 Stav RS na při zapnutí procesoru. [6]

B	Op1	V	Op2	V	RRN	Ctrl	R
0							
0							
0							
0							

Struktura rezervační stanice je uvedena v tabulce 2.2. R je bit připravenosti, který se nastavuje na 1 v případě, když oba bity validnosti jsou 1. R signalizuje, že instrukce je připravena k výdeji vykonávacím jednotkám. Alokace resp. vydaní realizované pomocí case-logiky a nejprve alokuje resp. vydá instrukci, která je v nejvyšší vhodné položce. Ve výpisu 2.5 je uvedena realizace case-logiky při vydaní instrukce v mojí implementaci v jazyce Verilog.

Výpis kódu 2.5 Realizace vydaní instrukcí z RS.

```
always@(posedge CLK) begin
        casez (ReadyFlow)
        4'b???1: ...
        4'b??10: ...
        4'b?100: ...
        4'b1000: ...
        default: ...
    endcase
    ...
end
```

Konstrukce `casez` zkoumá vektor `ReadyFlow`, který udržuje informaci o tom, které položky jsou připravené k vydaní. Nejnižší bit je první položka.

2.2.5 Výpočetní jednotky

Moje architektura implementuje 2 typy výpočetních jednotek. Rozdělil jsem je podle specializace. První typ implementuje operace sčítání a odčítání (instrukce ADD a SUB) a provádí výpočet za 1 takt. Druhý typ implementuje operace logického posunu (instrukce SRL a SLL) a provádí výpočet za 2 takty. Latence pro vykonaní jednotlivých instrukcí je zvolena pouze z ukázkových důvodů a neodpovídá HW realizaci ve skutečných mikroarchitekturách.

Výpočetní jednotka dostává 2 operandy, kód *ALUControl*, číslo skrytého registru a příznak, který udává, zda má ALU vyzvednout vstupní data a zahájit výpočty. ALU má také 2 vlastní příznaky. *Accepted* ukazuje, zda převzala vysílanou instrukci, aby RS mohla dealokovat příslušnou položku. *Calculated* ukazuje, zda ALU dokončila výpočet. Po výpočtu je výsledek odeslán ven spolu s číslem skrytého registru. Na základě hodnoty RRN ostatní komponenty jednoznačně identifikují výsledek a aktualizují příslušná pole.

2.2.6 Soubory registrů

Moje mikroarchitektura má 2 fyzicky oddělené soubory registrů. Obsahují 32 architekturálních (x0–x31) a 32 skrytých registrů. Rezervační stanice zvenku

vidí hodnoty a bity validnosti registrů. Schéma propojení registrů a rezervačních stanic bude podrobněji probraná později v kapitole věnované testovaní a simulaci.

2.2.6.1 Soubor architekturálních registrů

Tabulka 2.3 Vnitřní struktura souboru architekturálních registrů (skutečně soubor má 32 řádků). [6]

Data	V	RRN
	1	
	1	

Obsah architekturálních registrů je plně platný na začátku provádění programu. Jsou to registry, které programátor vidí a počítá s jejich hodnotami. Musí být aktualizovány v programovém pořadí.

Cílový registr nové instrukce se vždy nejprve přejmenuje a příslušný řádek v ARSet se zneplatní. Znovu se stává relevantním, až když dostane odpovídající signál z „in-order“ bufferu ROB. V jednom taktu může dojít i k aktualizaci několika položek. RRN je číslo skrytého registru, na něhož byla naposledy přejmenována instrukce s odpovídajícím architekturálním cílovém registrem. Vnitřní struktura ARSet je znázorněna v tabulce 2.3.

2.2.6.2 Soubor skrytých registrů

Skryté registry se přidělují podle stejné case-logiky jako ROB, tj. od nejvyššího k nejnižšímu. RRSet naslouchá spojení od ALU a aktualizuje odpovídající položky, pokud se RRN shodují. Do příštího přejmenování bit V je nastaven na 1. K dealokaci dojde, když je z ROB přijat odpovídající signál ve formě bitového vektoru, který ukazuje, které položky se musí zneplatnit. V takovém případě bit B je nastaven na 0 a položka je volná pro další alokaci. Vnitřní struktura RRSet je znázorněna v tabulce 2.4.

Tabulka 2.4 Vnitřní struktura souboru skrytých registrů (skutečně soubor má 32 řádků). [6]

Data	V	B
		0
		0

2.2.7 Specifičnosti propojení komponent CPU

V této podkapitole popíšeme součástky, které sami o sobě nemají roli v činnosti Tomasulova algoritmu ale slouží pro zabezpečení správného fungování ostatních komponent.

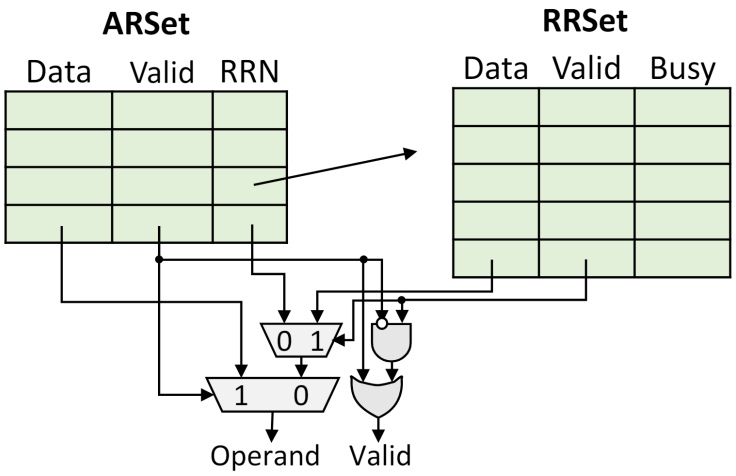
2.2.7.1 Synchronizace datových toků

Jednou z hlavních výzev při návrhu mé architektury se ukázala být synchronizace datových toků mezi komponentami. Protože procesor čte a vykonává 2 instrukce za takt, musel jsem správně určit pořadí, ve kterém komponenty vidí změny v souborech registrů.

Potíže vznikají zejména proto, že obě načtené instrukce přistupují k souborům registrů současně. A například v důsledku přejmenování cílového registru první načtené instrukcí se bit platnosti v ARSet změní na 0, ale druhá načtená instrukce tuto změnu nesleduje a čte starý nesprávný stav ARSet. Abych se tohoto a podobných problémů zbavil, zavedl jsem novou komponentu CLK-FirstInst. Jedná se o jednoduchý klopný obvod, který si pamatuje starý obsah souborů registrů. Na obrázku 2.1 je umístěn mezi ARSet a RRSet.

2.2.7.2 Načtení operandů do rezervačních stanic

Jedním z nejkomplikovanějších míst v mikroarchitektuře je načítání hodnot operandů a jejich validnosti do příslušných položek v rezervačních stanicích. Ve finální verzi jsem to realizoval pomocí série multiplexorů a pomocných obvodů. Na obrázku 2.2 uvádím realizaci, kterou jsem použil.



Obrázek 2.2 Kombinační logika načtení operandu do RS. [6]

Mohou nastat tři scénáře.

- Obsah architekturálního registru je validní a měli bychom číst data z něj.
- Obsah architekturálního registru není validní ale obsah skrytého registru je a měli bychom číst data z RRSet.
- Obsah obou registru platný není a měli bychom načíst číslo skrytého registru, do něhož se musí zapsat nutná hodnota.

Drát *Operand* předává buď data nebo číslo skrytého registru. Drát *Valid* předává informaci o tom, co jsme obdrželi. [6]

Navíc kombinační logiku komplikují okolnosti, o nichž jsem hovořil v podkapitole 2.2.7.1. Jako vstup lze použít data z „verze registrů z minulosti“. Také RS přijímá data, která byla právě vypočítána v výpočetních jednotkách v daném cyklu.

Kapitola 3

Simulační a testovací experimenty

Tato kapitola popisuje, jak jsem testoval implementaci. Uvádí příklady testovacích programů a vysvětluje, jak napsat vlastní testy a spustit je.

Mnoho oprav popsaných v kapitole 2.2.7 bylo v architektuře provedeno po testování dvěma základními testovacími programy. Než se jimi budeme podrobněji zabývat, rád bych upozornil na některé drobné změny v návrhu, které byly provedeny za účelem pohodlnějšího testování.

Při spuštění procesoru se provede reset a poté se z paměti načtou instrukce za sebou, délka taktu je 4 časové jednotky. Tato délka je zvolena z důvodu pohodlnějšího testování. Procesor pak začíná vykonávat instrukce od spuštění ihned po resetu. Poté už nikdy nepřestane číst data z instrukční paměti, dokud nedojde k vypnutí. Pro pohodlnější testování jsem předepsal v komponentě CrossSwitch umělý zámek program counteru, který se zapne v případě, že obě načtené instrukce berou operandy ze stejného registru `x4` a také je tam zapisují. Uvidíme ty 2 instrukce na konci každého testovacího programu.

Výpis kódu 3.1 Umělé pozastavení CPU

```
assign Stall =  RRSetFree < 2 || ROBFree < 2 ||
                RS1Free < 2   || RS2Free < 2 ||
                (SrcOp1[0] == 4 && SrcOp2[0] == 4 &&
                SrcTarget[0] == 4) ||
                (SrcOp1[1] == 4 && SrcOp2[1] == 4 &&
                SrcTarget[1] == 4);
```

Kód zámku (uvedeny ve výpisu 3.1) může být čtenářem zakomentován resp. změněn v souboru `src/CrossSwitch.v` na řádcích 78–82. Zakomentovaný úsek

je původně vymyšlený kód bez testovacích omezení.

Pro testování jsem také napsal samostatný modul v jazyce Verilog, který simuluje situaci, kdy procesor normálně běží v počítači. Výpis 3.2 uvádí always-cyklus, jenž před každým taktem vypíše stav a obsah všech potřebných proměnných.

Výpis kódu 3.2 Struktura testovacího modulu.

```
always begin
    $display ("Time: %t", $time);

    ... // dalsi potrebne vypisy

    clk <= 1;
    #2;
    clk <= 0;
    #2;
end
```

Pomocí tohoto modulu jsem vypisoval vnitřní stav registrů ve formě tabulky. Její příklady uvedu dále.

Na začátku jsem naplnil každý architekturální registr hodnotou 5.

3.0.1 Obě načtené instrukce sdílejí zásoby registrů

Nejprve jsem chtěl zjistit, jak procesor nakládá s prostředky registrů v situaci, kdy obě načtené instrukce čtení žádají o přístup ke stejným registrům. Jako testovací program jsem zvolil extrémní případ, kdy jsou data čtena a zapisována do stejného registru `x0` osmkrát za sebou. Program je uveden ve výpisu 3.3.

Výsledkem testů a následného ladění je komponenta CLKFirstInst, která ukládá předchozí stav. Díky tomu jsem dosáhl správného počítání dat. Konečný stav souboru architekturálních registrů je uveden v tabulce 3.1. Výsledek odpovídá předpokládanému $5 \cdot 2^8 = 1280$.

Tabulka 3.1 Výsledek práce CPU po prvním testovacím programu (skutečně soubor má 32 řádků).

	AllocationFlow	Data	V	RRN
ARSet[0]:	00	1280	1	3
ARSet[1]:	00	5	1	X
ARSet[2]:	00	5	1	X

Výpis kódu 3.3 První testovací program.

```
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x4, x4, x4
add x4, x4, x4
```

3.0.2 Složitější testy

Dále jsem architekturu testoval na programu, který zapojuje všechny druhy instrukcí, používá všechny vykonávací jednotky a rezervační stanice.

Výpis kódu 3.4 Druhý testovací program.

```
add x0, x1, x2
srl x3, x0, x2
sub x0, x1, x0
sll x1, x0, x3
add x2, x0, x0
sll x1, x0, x3
add x4, x4, x4
add x4, x4, x4
```

Po otestování a odladění všechny komponenty fungují správně. Stav architekturálních registrů je podle očekávání. Pořadí přidělování skrytých registrů z RRSet je správné.

Tabulka 3.2 Výsledek práce CPU po druhém testovacím programu (skutečně soubor má 32 řádků).

	AllocationFlow	Data	V	RRN
ARSet[0]:	00	4294967291	1	2
ARSet[1]:	00	4294967291	1	5
ARSet[2]:	00	4294967286	1	4
ARSet[3]:	00	0	1	1

3.0.3 Návod na provádění testů

Pokud si čtenář přeje provést vlastní testy, měl by postupovat podle následujících kroků. Je nutně to provádět na Linuxu s instalovaným balíčkem `iverilog`.

1. Přejít do složky se zdrojovými kódy.
2. Zkopírovat do něj dříve připravené testovací program ve formátu `.hex` a pojmenovat ho `memfile_inst.hex`.
3. Změnit výpis interních informací počet prováděných taktů v testovacím souboru `TestModule.v` nebo se podívat, jaké výchozí výpisy jsem v tomto souboru ponechal.
4. Zkompilovat spustitelný soubor pomocí příkazu `iverilog *.v -g2012`.
5. Spustit spustitelný soubor příkazem `./a.out` a prostudovat výpis konzole.

Pokud by čtenář chtěl spustit mé testy z práce, v kroku 2 by měl si zkopírovat jejich zdrojový kód ve formátu `.hex` ze složky `test programs`. Soubory se jmenují `firstTest.hex` a `secondTest.hex`.

Kapitola 4

Závěr

Výsledkem této práce je návrh a implementace vlastní mikroarchitektury superskalárního procesoru, která umožňuje vykonávat určitou podmnožinu instrukcí RISC-V RV32I. Implementace je napsána v jazyce Verilog a testována pomocí nástroje Icarus Verilog. V teoretické části jsme se seznámili s typy procesorů z hlediska paralelního vykonávání instrukcí, seznámili jsme se se základními principy mikroarchitektur, které vykonávají instrukce mimo programové pořadí, a podívali jsme se na příklady kódování assemblerovských instrukcí v ISA, která se používají i dodnes.

V praktické části jsem popsal svou realizaci daných principů. V mém návrhu je prostor pro zlepšení. Implementoval jsem pouze instrukce typu Registr-Registr. Bylo by možné přidat skokové instrukce a na jejich příkladu ukázat spekulativní provádění různých větví instrukcí. Nebo na příkladu instrukcí pracujících s pamětí by bylo možné ukázat technologie efektivního zápisu do paměti, práci s cache.

Bibliografie

1. JEŽEK, Pavel. Principy počítačů a operačních systémů. In: *Zvyšování výkonnosti procesorů* [online]. School of Computer Science, Faculty of Mathematics a Physics, Charles University, 2011 [cit. 2024-05-05]. Dostupné z: `https://d3s.mff.cuni.cz/legacy/teaching/principles_of_computers/ar-20112012/06-zrychlovani.pdf`.
2. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. Architektura souboru instrukcí (ISA) a výkonnost počítačů. In: *BI-APS Architektury počítačových systémů* [online]. Czech Technical University in Prague, Faculty of Information Technology, 2023 [cit. 2024-04-28]. Dostupné z: `https://courses.fit.cvut.cz/BI-APS/media/lectures/BI-APS-Prednaska01-PerfEvaluation-ISA.pdf`.
3. WATERMAN, Andrew; ASANOVIĆ, Krste (ed.). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. In: *RISC-V* [online]. RISC-V Foundation, 2017 [cit. 2024-04-28]. Dostupné z: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`.
4. CHESALOV, Alexander. *The fourth industrial revolution glossarium: over 1500 of the hottest terms you will use to create the future* [online]. books.google.cz, 2023 [cit. 2024-04-29]. Dostupné z: `https://books.google.cz/books?id=VlG5EAAAQBAJ&dq=cpu+electronic+circuitry+executes+instructions+of+a+computer+program,+such+as+arithmetic,+logic,+controlling,+and+input/output+(I/O)+operations&pg=PT54&redir_esc=y#v=onepage&q=cpu%20electronic%20circuitry%20executes%20instructions%20of%20a%20computer%20program%2C%20such%20as%20arithmetic%2C%20logic%2C%20controlling%2C%20and%20input%2Foutput%20(I%2FO)%20operations&f=false`.
5. VĚŽNÍK, Tomáš. *RISC-V CPU superscalar microarchitecture design* [online]. Czech Technical University in Prague, Faculty of Information Technology, 2022 [cit. 2024-04-29]. Dostupné z: `https://projects.fit.cvut.cz/theses/4422`. Dis. pr.

6. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. Superskalární procesory I: Úvod. In: *BI-APS Architektury počítačových systémů* [online]. Czech Technical University in Prague, Faculty of Information Technology, 2023 [cit. 2024-04-28]. Dostupné z: `https://courses.fit.cvut.cz/BI-APS/media/lectures/BI-APS-Prednaska10-SuperscalarCPUs-I.pdf`.
7. Lecture 8: Issues in Out-of-order Execution. In: *Computer Architecture* [online]. Carnegie Mellon University, 2010 [cit. 2024-05-05]. Dostupné z: `https://course.ece.cmu.edu/~ece740/f10/lib/exe/fetch.php?media=740-fall10-lecture8-afterlecture-issues_in_ooo.pdf`.
8. TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*. 1967, roč. 11, č. 1, s. 25–33. Dostupné z DOI: `10.1147/rd.111.0025`.
9. Description of Tomasulo Based Machine. In: *University of California San Diego* [online]. University of California San Diego, 2007 [cit. 2024-05-04]. Dostupné z: `https://cseweb.ucsd.edu/classes/fa07/cse240a/Papers/tomasulo.pdf`.

Obsah přiložených souborů

- `out` ..... adresář se spustitelnou formou implementace
  - `memfile_inst.hex` ..... zdrojový kód prvního testovacího programu
  - `a.out` ..... spustitelný soubor
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `test programs` ..... složka s testovacími programy
      - `firstTest.hex`
      - `secondTest.hex`
    - `TestModule.v` ..... testovací modul
    - `alusSuper.v`
    - `CPUmainSuper.v`
    - `CrossSwitch.v`
    - `GPRSetSuper.v`
    - `Reservation.v`
    - `ROB.v`
    - `Sets.v`
    - `TopTest.v`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `Media` ..... složka s použitými v práci obrázky
    - `text` ..... složka s textem práce a bibliografií
    - `ctufit-thesis.cls`
    - `ctufit-thesis.tex`
    - `egoroale-assignment.pdf`
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF