Masarykova univerzita
Fakulta informatiky

Návrh modelu procesoru

Bakalářská práce

Dominik Salvet

Brno, jaro 2020

Masarykova univerzita
Fakulta informatiky

Návrh modelu procesoru

Bakalářská práce

Dominik Salvet

Brno, jaro 2020

Na tomto místě se v tištěné práci nachází oficiální podepsané zadání práce a prohlášení autora školního díla.

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Dominik Salvet

Vedoucí práce: prof. Ing. Václav Přenosil, CSc.

Poděkování

Tímto bych chtěl vyjádřit velké poděkování vedoucímu mé bakalářské práce prof. Ing. Václavu Přenosilovi, CSc. za cenné rady nejen v rozsahu této práce. Zvláště bych chtěl poděkovat za projevenou trpělivost, čehož si velmi vážím.

Děkuji také mé rodině za neustálou podporu a důvěru.

Shrnutí

Cílem této práce je navrhnout model 64bitového procesoru vlastní instrukční sady a doložit jeho funkčnost. Tato instrukční sada respektuje RISC principy při snaze zvýšit hustotu výsledného kódu. Za tímto účelem definuje výhradně instrukční slova s délkou 16 bitů.

Referenční model implementující tuto instrukční sadu je popsán v jazyce VHDL pomocí základního pětistupňového pipeliningu. Lze ho simulovat pomocí programu GHDL a, s malým úsilím, implementovat do hradlového pole dostatečné kapacity. Správné chování procesoru a jeho stěžejních modulů je doloženo pomocí test bench souborů.

Klíčová slova

procesor, RISC, ISA, pipeline, VHDL, ALU, testbench, přerušení

Obsah

Úvod **1**

1 Vlastnosti instrukční sady **3**
1.1 *Typ instrukční sady* . . . 3
1.1.1 CISC . . . 4
1.1.2 RISC . . . 4
1.2 *Bitová šířka dat* . . . 5
1.3 *Bitová šířka instrukcí* . . . 5
1.4 *Architekturální registry* . . . 6
1.5 *Třída instrukční sady* . . . 7
1.6 *Organizace paměti* . . . 7

2 Návrh instrukční sady **9**
2.1 *Volba vlastností instrukční sady* . . . 9
2.2 *Řídící registry* . . . 10
2.2.1 Pomocné řídící registry (`k0`, `k1`) . . . 11
2.2.2 Status registr (`status`) . . . 11
2.2.3 Registr vektoru přerušení (`ivec`) . . . 12
2.2.4 Uložený čítač instrukcí (`spc`) . . . 12
2.3 *Formáty instrukcí* . . . 12
2.3.1 Instrukce pro komunikaci s pamětí . . . 13
2.3.2 Instrukce přičítání konstant . . . 14
2.3.3 Instrukce podmíněných skoků . . . 14
2.3.4 Aritmetické instrukce s konstantou . . . 15
2.3.5 Aritmeticko-logické instrukce . . . 16
2.3.6 Nepodmíněná skoková instrukce . . . 17
2.3.7 Instrukce pro manipulaci s daty . . . 17
2.3.8 Instrukce pro řídící registry . . . 18
2.3.9 Instrukce datového přesunu . . . 19
2.3.10 Instrukce bez operandů . . . 19

3 Návrh mikroarchitektury **21**
3.1 *Popis pipeline* . . . 21
3.1.1 Fáze vyzvednutí instrukce (IF) . . . 22
3.1.2 Fáze dekódování instrukce (ID) . . . 23

3.1.3 Fáze provádění instrukce (EX) 24
3.1.4 Fáze přístupu do paměti (MEM) 25
3.1.5 Fáze zápisu výsledku (WB) 26
3.1.6 Pipeline hazardy 26
3.2 *Rozhraní pro komunikaci s pamětí* 27
3.2.1 Rozhraní pro instrukční paměť 27
3.2.2 Rozhraní pro datovou paměť 28
3.3 *Úvod do VHDL projektu* 28
3.3.1 Použité konvence 28
3.3.2 Adresářová struktura 29

4 Simulace mikroarchitektury **31**
4.1 *GHDL simulátor* 32
4.2 *Vlastní build systém* 32

Závěr **35**

Bibliografie **37**

A Elektronické přílohy **39**

B Schéma pipeline **41**

Úvod

Návrh procesoru je obecně velmi rozsáhlá činnost několika vědních oborů vyžadující pokročilé znalosti číslicové techniky, hardwarových principů, principů nízkoúrovňového softwaru i specifických oblastí fyziky. Návrh modelu procesoru je podmnožinou této činnosti a omezuje se pouze na návrh logické struktury procesoru, příp. i tvorby jeho virtuální implementace.

Tato práce představuje model 64bitového RISC[1] procesoru vlastní architektury a popisuje jeho funkce. K vytvoření takového modelu je nejdříve definována vlastní architektura instrukční sady (dále jen ISA[2]), která používá menší šířku instrukčního slova, než je běžné u architektur s podobnými rysy. To zpravidla vede k nižším paměťovým nárokům většiny programů, což se nakonec může pozitivně projevit i ve výkonu procesoru [1]. Tímto přístupem se totiž snižuje tlak na instrukční paměť, která je v moderních procesorech[3] implementována jako instrukční cache a významně se podílí na celkovém výkonu procesoru. Hlavní motivací práce je tedy prozkoumat dopad menších instrukcí na samotný návrh modelu procesoru (dále už jen procesor) a jeho architekturu.

Součástí práce je i referenční mikroarchitektura, která zmíněnou ISA kompletně implementuje. Je popsána v jazyce VHDL metodou pětistupňového pipeliningu a podporuje i okamžitá přerušení. Hlavní modul procesoru vystavuje jednoduché rozhraní pro komunikaci s instrukční a datovou pamětí. Pakliže jsou tyto paměťové moduly dostupné, je procesor připraven na umístění do hradlového pole a demonstraci jeho možností na skutečném hardwaru.

V neposlední řadě je práce doprovázena i test bench soubory, které ověřují funkčnost procesoru i jeho klíčových komponent. K simulaci procesoru na počítači se používá open-source program GHDL[4] a pro snadnější práci s celým build systémem procesoru byl vytvořen i jednoduchý `makefile`.

---

1. Reduced Instruction Set Computer – počítač s redukovanou instrukční sadou
2. Z anglického překladu Instruction Set Architecture
3. Moderním procesorem rozumíme procesor pro osobní počítače.
4. G Hardware Design Language – G jazyk pro návrh hardwaru

Samotný text práce je rozčleněn do několika dílčích kapitol. První kapitola se zabývá ustanovením základních vlastností ISA. Na základě definic z první kapitoly se druhá kapitola zabývá návrhem ISA a zmiňuje i významné procesorové architektury, které měly vliv na její vývoj. Ve třetí kapitole je tato navržená ISA implementována popisem v jazyce VHDL. Zahrnuje i detailní vhled do mikroarchitektury procesoru. Praktičtější povahu má kapitola čtvrtá, která ukazuje, jak obsluhovat build systém procesoru. Důraz je kladen zejména na simulaci procesoru pro účely testování.

Model procesoru vzniknuvší v rámci této práce nese označení RISC63 a jeho zdrojové kódy a ostatní materiály jsou veřejně dostupné z GitHub repositáře[5] pod Apache 2.0 licencí.

5. `https://github.com/dominiksalvet/risc63`

1 Vlastnosti instrukční sady

Instrukční sada je abstraktní soubor pravidel, který jednoznačně definuje rozsah funkcionality procesoru. Definuje tedy jednotlivé strojové instrukce a jejich sémantiku, architekturální registry, organizaci paměti, mechanismus přerušení a další. Tyto vlastnosti mají přímý vliv na výslednou implementaci procesoru. Pro programátora, resp. kompilátor, se pak jedná o dostupné prostředky procesoru implementujícího danou ISA.

V této kapitole jsou popsány významné vlastnosti ISA v kontextu této práce. Asi nejznámější vlastností ISA je její typ. Ačkoliv přesná definice tohoto pojmu neexistuje, je široce užíván ve snaze implicitně vystihnout dílčí vlastnosti ISA. Další zkoumané vlastnosti ISA pak zahrnují bitovou šířku dat, bitovou šířku instrukcí, architekturální registry, třídu instrukční sady a organizaci paměti. Na základě těchto definic jsme pak schopni v další kapitole odvodit možné dostupné instrukce ISA a jejich vlastnosti.

Každá podkapitola se věnuje pouze jedné z uvedených vlastností. Popíše danou vlastnost, jak se obvykle volí její parametry, možný dopad na architekturu procesoru a uvede i soudobé populární hodnoty parametrů (rok 2020).

1.1 Typ instrukční sady

Typem instrukční sady rozumíme pro účely této práce klasifikaci podle složitosti procesorové architektury. Přestože se nejedná o žádný přesně definovaný postup, je daná ISA podle převažujících rysů zařazena do nejvhodnějšího, zpravidla již existujícího, typu ISA. Velmi vzácně se však může stát, že nová ISA je natolik specifická, aby dala vzniknout novému typu ISA, který ji vystihuje [2].

Pro potřeby této práce se v níže uvedené analýze omezíme jen na dva nejrozšířenější typy ISA, kterými jsou CISC[1] a RISC.

1. Complex Instruction Set Computer – počítač s komplexní instrukční sadou

1.1.1 CISC

CISC lze považovat za nejstarší ustanovený typ ISA. Nejvíce jej zviditelnila firma Intel se svou CISC architekturou x86, která byla později nahrazena kompatibilní CISC architekturou AMD64. CISC se vyznačuje především obsáhlým repertoárem instrukcí. Obecně také definuje proměnlivou délku těchto instrukcí, nižší počet pracovních registrů a více podporovaných adresních módů pro přístup k paměti.

Lze konstatovat, že tento typ ISA je na ústupu, protože její výhody spojené s nízkými paměťovými nároky programů již nejsou tak důležité, jako tomu bylo dříve. Navíc dekodér CISC instrukcí v celé řadě případů limituje efektivní implementaci moderních mechanismů pro zvýšení výkonu procesoru. Toto tvrzení podporuje i skutečnost, že x86/AMD64 je nyní jedinou rozšířenou CISC architekturou.

1.1.2 RISC

Vznik RISC architektury lze připsat snaze zvýšit výkon tehdejších CISC procesorů, které byly poměrně složité. Zároveň se tehdy snižovala cena operační paměti. Tyto okolnosti vyústily v ustanovení nového typu ISA, jehož hlavním pilířem je jednoduchost výsledné mikroarchitektury. Z toho důvodu RISC procesory obsahují jen jednoduché a nejpoužívanější instrukce, mají vyšší počet pracovních registrů (obecného použití) a podporují pouze několik adresních módu.

Jednodušší RISC architektury definují instrukce s pevnou délkou. Pokročilejší RISC architektury většinou definují vedle toho i kratší instrukce, označovány jako komprimované, které mohou představovat aliasy pro nejpoužívanější instrukce plné velikosti [3, s. 97–113]. Motivací k tomuto přístupu je zvýšit hustotu kódu, což vede ke snížení tlaku na instrukční cache, a to pak zvyšuje výkon procesoru.

Dnešní RISC architektury dosahují hustoty kódu, která je srovnatelná nebo dokonce lepší než hustota kódu CISC architektur [4]. Stále přitom nabízí výhody spojené s jednoduchostí RISC návrhu. Obecně se pak dá tvrdit, že přízeň k RISC architekturám roste od doby jejich vzniku. Mnoho aktuálních a dokonce i nově vznikajících architektur je právě typu RISC.[2]

2. Viz např. `https://riscv.org`.

1.2 Bitová šířka dat

Bitová šířka dat, též označovaná jako bitová šířka slova, představuje největší počet bitů dat, se kterými dokáže ISA nativně pracovat. Při práci s daty vyšší bitové šířky je zapotřebí implementovat požadovanou operaci pomocí několika dílčích kroků.

Hodnota bitové šířky dat má klíčový význam pro mnoho vlastností procesoru. Přímo ovlivňuje bitovou šířku aritmeticko-logické jednotky (dále jen ALU[3]) a pracovních registrů. Velmi často pak ovlivňuje i architekturu jiných modulů procesoru a bitové šířky sběrnic, se kterými procesor pracuje. Může se jednat o architekturu cache a bitové šířky datové[4] i adresní sběrnice procesoru.

Menší hodnoty bitové šířky dat obecně znamenají rychlejší propagaci dat skrze procesor a nižší požadavky na hardwarové prostředky. Větší hodnoty bitové šířky dat ovšem umožňují procesoru efektivněji pracovat s daty vyšších bitových šířek a adresovat větší paměťový prostor. Volba bitové šířky dat se zejména odvozuje z potřebné kapacity paměťového prostoru, který má být schopen procesor efektivně adresovat. Zvýšit kapacitu adresovatelného paměťového prostoru již existující procesorové architektury bývá totiž netriviální úkon.

Nejčastější šířka dat v moderních procesorech je 64 bitů a zřídka se ještě používají procesory s šířkou dat 32 bitů. Takové procesory jsou souhrnně označovány jako 64bitové, resp. 32bitové. Důvodem vzestupu popularity 64bitové šířky dat byly zejména zvyšující se paměťové požadavky softwaru a s tím spjatá potřeba navýšení kapacity paměťového prostoru adresovatelného procesorem. Tento trend byl doprovázen také postupnou transformací 32bitového softwaru na 64bitový, který je schopen možností 64bitového procesoru využít.

1.3 Bitová šířka instrukcí

Bitová šířka instrukcí, též označovaná jako bitová délka instrukčního slova, představuje počet bitů instrukčního slova dané ISA. Tento pojem však není široce zavedený, protože existují architektury, které mají proměnlivou délku instrukce. Jednoznačně jej lze tedy apliko-

3. Z anglického překladu Arithmetic Logic Unit
4. Tato sběrnice často udává konečnou bitovou šířku přenosu dat.

vat jen na architektury s pevnou délkou instrukcí. Z hodnoty bitové šířky instrukcí lze přímo odvodit celkový počet unikátních instrukcí vykonatelných procesorem.

Při volbě hodnoty bitové šířky instrukcí je nutné si uvědomit, že mnoho instrukcí bude obsahovat bity reprezentující indexy pro pracovní registry procesoru nebo bity reprezentující konstanty. A proto větší hodnoty bitové šířky instrukcí znamenají zejména vyšší počet přímo adresovatelných pracovních registrů a větší bitovou šířku konstant zabudovaných přímo v instrukcích. Naopak menší šířka instrukcí (do určité míry) znamená nižší paměťové požadavky výsledného softwaru [1]. Pokud je navíc bitová šířka instrukcí menší než bitová šířka přenosu dat, lze provádět přenos více instrukcí z operační paměti v rámci jednoho čtení přímo na úrovni architektury procesoru.[5]

Nejčastější šířka instrukcí u dnešních procesorů s pevnou délkou instrukcí je 32 bitů. Ovšem jak již bylo zmíněno, některé architektury umí provádět i komprimované instrukce, které mívají zpravidla 16 bitů. Tyto hodnoty reprezentují kompromis mezi snadným dekódováním instrukcí a výslednou hustotou kódu. Na tomto místě je pravděpodobně vhodné upozornit, že šířka instrukcí a šířka dat nemusí být, a většinou nebývá, rovna.

1.4 Architekturální registry

Architekturální registry (dále jen AR) jsou registry, které definuje samotná ISA. Reprezentují stav vykonávaného programu a často i konfiguraci procesoru. Nejčastějším typem AR je PC[6], který udává adresu právě vykonávané instrukce. Dalšími AR jsou pracovní registry a mnohdy i registry řídící, které se podílí např. na správě systému přerušení daného procesoru.

Pracovní registry jsou klíčovými registry pro práci s daty uvnitř procesoru. Programy je totiž mohou přímo využívat pro realizaci svého výpočtu. Pokud mohou být mezi sebou libovolně zaměňovány, hovoříme o pracovních registrech obecného použití. Jejich počet může být limitován délkou instrukčního slova a příp. cílovým zaměřením procesoru. Nejčastější počet pracovních registrů v moderních proce-

5. Obdobně lze toto tvrzení aplikovat i na cache procesoru.
6. Program Counter – čítač instrukcí

sorech je 32, a 16 registrů mívají procesory v embedded oblasti. Je důležité si uvědomit, že se nejedná o celkový počet registrů mikroarchitektury, který bývá zpravidla mnohem vyšší.

1.5 Třída instrukční sady

Třída instrukční sady udává, jakým způsobem jsou organizovány pracovní registry procesoru, a ovlivňuje způsob vykonávání instrukcí nad nimi. Její nejdůležitější atribut je počet dostupných operandů ve většině instrukcí, a tak je volba třídy ISA efektivně limitována volbou šířky instrukcí. Podle počtu operandů jsou níže uvedené významné třídy ISA:[7]

- Nula – zásobníkový počítač
- Jeden – akumulátorový počítač
- Dva – počítač registr-registr
- Tři – počítač registr-registr-registr

Obecně platí, že více operandů implikuje více příležitostí pro zvýšení výkonu výsledné mikroarchitektury. I z toho důvodu prakticky všechny moderní procesory umí pracovat se třemi operandy najednou. U RISC procesorů se jedná o již zmíněnou třídu registr-registr-registr, která je též označována jako load-store architektura.

1.6 Organizace paměti

Organizace paměti dané ISA představuje množinu pravidel a dostupných funkcí k obsluze operační paměti. To pak implikuje maximální adresovatelnou kapacitu paměti, nejmenší adresovatelnou jednotku a endianitu[8] paměti i možné velikosti přenášených bloků dat mezi procesorem a pamětí. Způsob organizace paměti má tedy klíčový význam pro všechny programy běžící na procesoru.

7. Zejména se pohybujeme v oblasti RISC architektur.
8. Pořadí bytů větších datových bloků, v jakém jsou uloženy v operační paměti.

Ačkoli je dnes většina moderních procesorů 64bitových, z důvodu využívání stránkování paměti nedokáží adresovat celý 64bitový adresní prostor. Většinou dovedou adresovat prostor o velikosti 2^{48} bytů. Vlastností, která spojuje prakticky všechny dostupné procesory současnosti, je velikost nejmenší adresovatelné jednotky jeden byte (8 bitů). Většina procesorů umí přenášet i násobky bytu při komunikaci s pamětí. Existovaly a existují však i takové procesory, které používají byte jako nejmenší adresovatelnou jednotku, ale přenos jednotlivých bytů neumí [5]. K uložení dat v operační paměti se nejčastěji používá přístup little-endian, který respektuje přirozené pořadí bytů uvnitř větších datových bloků.

2 Návrh instrukční sady

V této kapitole je představena vlastní instrukční sada, která nese označení RISC63. Původně se jednalo o prototyp instrukční sady RISC64, která měla být plnohodnotnou instrukční sadou pro běh moderních operačních systémů. Implementace RISC63 však poukázala na její nedostatky, které zapříčinily zastavení dalšího vývoje RISC64 pro nevhodnost ISA ke zmíněnému účelu. RISC63 však žádné prvky pro podporu běhu operačního systému neimplementuje, a tak lze na tuto ISA pohlížet jako na procesorovou architekturu nezávislou. Bez nějakých obtíží lze tedy naplnit původní motivaci této práce, a to analyzovat dopady menšího instrukčního slova na architekturu procesoru.

2.1 Volba vlastností instrukční sady

RISC63, jak již z názvu vyplývá, je navržena jako RISC architektura a důsledně následuje její principy. Tento typ ISA byl zvolen zejména pro svou jednoduchost a popularitu v posledních letech.

Šířka dat RISC63 je 64 bitů. Tuto architekturu tedy označujeme jako 64bitovou. K rozhodnutí navrhnout RISC63 architekturu jako 64bitovou vedly následující skutečnosti. V současnosti se jedná o nejvyšší rozumnou[1] a obecně využitelnou hodnotu používanou k adresaci operační paměti. A jelikož RISC63 umožňuje adresovat operační paměť pomocí hodnoty pracovního registru, je výhodné zvolit šířku dat právě 64 bitů.

Zvláštností celé RISC63 architektury je pak šířka instrukčního slova. Používají se zde totiž 16bitové instrukce. V kombinaci s bitovou šířkou dat 64 lze hovořit téměř o unikátnosti architektury. Volba 16bitových instrukcí byla vnímána jako potencionální způsob, jak zvýšit výkon moderních RISC procesorů. Ačkoli se tato práce nezabývá obecným zhodnocením výkonu architektur s 16bitovými instrukcemi, uvádí poznatky z návrhu architektury RISC63, které určitou informační hodnotu v této oblasti poskytují. Při přenosu instrukcí z paměti jsou jejich adresy automaticky přirozeně zarovnávány směrem dolů.

1. Mocnina čísla 2

RISC63 definuje 64bitový architekturální registr PC použitý standardním způsobem jako ukazatel na aktuálně vykonávanou instrukci. Po restartu procesoru má tento registr hodnotu nula. Také definuje 16 pracovních registrů obecného použití, každý o velikosti 64 bitů. Jsou označeny jako `r0` až `r15`. Jejich hodnota po restartu procesoru není definovaná. Standardní praxí RISC architektur je zvolit jeden pracovní registr, který bude obsahovat vždy hodnotu nula.[2] To potom vede k řadě optimalizací a zjednodušení architektury procesoru. Tento postup však nemá uplatnění v architektuře RISC63, a proto jsou všechny pracovní registry zcela identické. Příčinou je použitá třída instrukční sady.

RISC63 používá instrukce se dvěma operandy typu registr-registr. Zvažovány byly i instrukce se třemi operandy typu registr-registr-registr, ale to by nevyhnutelně vedlo ke snížení počtu pracovních registrů na 8 a výrazným způsobem by to i zmenšilo dostupný instrukční prostor. Většina RISC63 instrukcí tedy čte hodnoty dvou pracovních registrů, přičemž do jednoho z nich pak zapisuje výsledek. Jedná se o výrazný ústupek způsobený 16bitovými instrukcemi.

RISC63 dokáže adresovat celý 64bitový adresní prostor a podporuje výhradně přirozeně zarovnané 64bitové datové přenosy mezi procesorem a operační pamětí. V případě nezarovnaného přístupu se adresa automaticky zarovná směrem dolů. Vzhledem k tomu, že RISC63 adresuje paměť po bytech, definuje i podpůrné instrukce pro manipulaci s datovými bloky menších velikostí než 64 bitů (viz kapitola 2.3.7). Tento přístup je inspirován architekturou prvních procesorů DEC[3] Alpha. Pro uložení dat v paměti se používá little-endian metoda.

2.2 Řídící registry

Architektura RISC63 podporuje vnější přerušení procesoru. Vnějším přerušením procesoru rozumíme externí asynchronní událost vyžadující bezprostřední reakci procesoru. Procesor je o tomto přerušení informován signálem žádosti o přerušení. Pro účely správy mechanismu přerušení definuje RISC63 architektura pět řídících registrů. Tyto re-

2. Zápis do takového registru je ignorován.
3. Digital Equipment Corporation

gistry mají architekturální velikost 64 bitů a v rámci optimalizace při zachování stejného chování mohou mít na úrovni mikroarchitektury skutečnou velikost menší. Pokud není uvedeno jinak, jejich hodnota po restartu procesoru není definovaná. Pro čtení a zápis řídících registrů jsou později představeny speciální instrukce (viz kapitola 2.3.8). Tento mechanismus řízení přerušení se vesměs inspiroval od MIPS[4] architektury.

2.2.1 Pomocné řídící registry (`k0`, `k1`)

Registry `k0` a `k1` nemají žádné hardwarově dedikované použití. Očekává se, že budou použity v rámci obsluhy přerušení. V momentě, kdy procesor akceptuje přerušení, tak vstoupí do rutiny obsluhy přerušení. V pracovních registrech jsou ovšem stále data přerušeného programu. Do jednoho pomocného registru se tedy uloží hodnota zvoleného pracovního registru a v druhém pomocném registru byl již předem uložen ukazatel na zásobník, který se načte do stejného pracovního registru. Potřebný počet zbylých registrů se pak pohodlně uloží na zásobník. Pokud obsluha přerušení potřebuje jen dva pracovní registry, může je uložit přímo do pomocných registrů. Před návratem z přerušení se musí provést opačný postup.

2.2.2 Status registr (`status`)

Registr `status` má centrálně uchovávat status a konfiguraci celého procesoru RISC63 architektury. Momentálně však definuje pouze LSB[5] jako IE[6] bit, který má po restartu procesoru hodnotu nula. Před povolením přerušení po restartu procesoru je třeba ještě nastavit vektor přerušení (viz kapitola 2.2.3).

Na rozdíl od pomocných registrů, `status` registr není modifikován pouze instrukcí pro zápis do řídících registrů. Pokud totiž přijde žádost o přerušení a přerušení je povoleno hodnotou jedna v IE bitu, přerušení je akceptováno a bit IE je automaticky nastaven na nulu. Zabrání se tak okamžitému vnořenému přerušení, na které rutina

4. Microprocessor without Interlocked Pipelined Stages – procesor bez automaticky řízené pipeline
5. Least Significant Bit – nejméně významný bit
6. Interrupt Enable – povolení přerušení

přerušení ještě nemohla procesor připravit. RISC63 nedefinuje žádný signál potvrzení přijetí přerušení ani řadič přerušení, a tak se o zrušení signálu žádosti o přerušení musí postarat sama rutina přerušení ještě před návratem.

Při návratu z přerušení použitím speciální instrukce určené pro tento účel se automaticky hodnota IE bitu nastaví na jedničku. Kdyby tomu tak nebylo, neexistoval by v RISC63 způsob korektního návratu z obsluhy přerušení. Cílem je navíc vrátit procesor do původního stavu před přerušením. A pokud se navrací z obsluhy přerušení, pak do ní mohl vstoupit jedině, pokud bylo přerušení předtím povoleno.

2.2.3 Registr vektoru přerušení (`ivec`)

Registr `ivec` obsahuje adresu, na kterou se skočí při přijetí přerušení. Na této adrese se musí nacházet rutina obsluhy přerušení.

2.2.4 Uložený čítač instrukcí (`spc`)

Jakmile dojde k přijetí přerušení, další potenciální adresa PC registru je zapsána do registru `spc`. Při návratu z přerušení se pak hodnota `spc` automaticky přesune do registru PC, což efektivně způsobí transparentní návrat do přerušeného programu.

2.3 Formáty instrukcí

Pro efektivnější implementaci instrukčního dekodéru jsou instrukce v RISC63 rozděleny do jednotlivých formátů, které sdílí některé důležité rysy. Nejedná se však o velmi efektivní rozdělení instrukcí podle pevně určených bitů instrukčního slova tak, jak lze pozorovat u typických RISC architektur. Finální instrukční formát je spíše daný variabilním počtem bitů vyššího bytu instrukčního slova. Jedná se o omezení, které je úzce spjaté s 16bitovou šířkou instrukčního slova.

Následující část práce popisuje všechny instrukční formáty RISC63 architektury. Z důvodu přehlednosti jsou k tomuto účelu intenzivně využívány tabulky zachycující přesnou strukturu individuálních instrukcí a zde definované symboly:

- *func* – konkrétní funkce daného instrukčního formátu
- *imm* – konstanta rozšířená podle jejího znaménkového bitu
- *rb* – index čteného pracovního registru
- *ra* – index čteného i zapisovaného pracovního registru
- *cr* – index řídícího registru
- *x* – bity nejsou v daném instrukčním formátu použity

Způsob níže uvedeného zobrazení a popisu instrukcí se inspiruje dokumentací instrukční sady RISC-V [3].

2.3.1 Instrukce pro komunikaci s pamětí

Tabulka 2.1: Struktura instrukcí pro komunikaci s pamětí

15	14	13	12 … 8	7 … 4	3 … 0
1	1	func	imm	rb	ra
2		1	5	4	4
		0 (LD)			pouze zápis
		1 (ST)			pouze čtení

LD načte 64bitovou hodnotu z paměti do registru *ra*. ST uloží 64bitovou hodnotu z registru *ra* do paměti. Adresa je vypočítána jako $rb + 8 \cdot imm$.

Hodnota imm *se posouvá před součtem o tři bity doleva, protože při adresování paměti dat jsou tři spodní bity nulovány z důvodu zarovnaného přístupu. Efektivně je tedy 8bitová.*

2.3.2 Instrukce přičítání konstant

Tabulka 2.2: Struktura instrukcí přičítání konstant

15	14	13	12	11	4	3	0
1	0	func		imm		ra	
2		2		8		4	
		00 (ADDI)					
		01 (ADDUI)					
		10 (AUIPC)				pouze zápis	
		11 (LI)				pouze zápis	

ADDI přičte do registru *ra* hodnotu *imm*. ADDUI přičítá hodnotu $256 \cdot imm$. AUIPC přičte k adrese instrukce hodnotu $256 \cdot imm$ a uloží ji do registru *ra*. LI načte hodnotu *imm* do registru *ra*.

ADDUI má široké použití při práci s 16bitovými konstantami. Nejprve modifikuje vyšší byte a další instrukce jen přičte offset v rámci spodního bytu.

2.3.3 Instrukce podmíněných skoků

Tabulka 2.3: Struktura instrukcí podmíněných skoků

15	13	12	11	10	4	3	0
0	1 1	func		imm		ra	
3		2		7		4	
		00 (JZ)				pouze čtení	
		01 (JNZ)				pouze čtení	
		10 (AIPC)				pouze zápis	
		11 (JR)				pouze čtení	

JZ a JNZ skočí na relativní adresu v rozsahu $2 \cdot imm$ vzhledem k adrese instrukce, pokud je hodnota *ra* rovna, resp. nerovna nule. AIPC přičte

k adrese instrukce hodnotu $2 \cdot imm$ a uloží ji do registru *ra*. JR skočí na adresu $ra + 2 \cdot imm$.

Hodnota imm *se posouvá před součtem o jeden bit doleva, protože při adresování paměti instrukcí je spodní bit nulován z důvodu zarovnaného přístupu. Efektivně je tedy 8bitová.*

RISC63 nedefinuje instrukci volání funkce. Namísto toho se použije AIPC pro uložení návratové adresy a pak se provede skok do dané funkce. Při návratu z funkce se použije JR.

2.3.4 Aritmetické instrukce s konstantou

Tabulka 2.4: Struktura aritmetických instrukcí s konstantou

15 13	12 10	9 4	3 0
0 1 0	func	imm	ra
3	3	6	4
	000 (SLTI)		
	001 (SLTUI)		
	010 (SGTI)		
	011 (SGTUI)		
	100 (SRLI)		
	101 (SLLI)		
	110 (SRAI)		
	111 (RSBI)		

SLTI a SGTI uloží do registru *ra* hodnotu jedna, pokud platí $ra < imm$, resp. $ra > imm$. V opačném případě uloží nulu. SLTUI a SGTUI jsou obdobnými variantami, přičemž hodnoty *ra* a *imm* interpretují jako neznaménkové.[7] SRLI a SLLI provádí logický bitový posun registru *ra* o *imm* mod 64 bitů doprava, resp. doleva. SRAI provádí aritmetický bitový posun doprava. RSBI uloží do registru *ra* hodnotu $imm - ra$.

7. Hodnota *imm* je však stále rozšířena podle jejího znaménkového bitu

2.3.5 Aritmeticko-logické instrukce

Tabulka 2.5: Struktura aritmeticko-logických instrukcí

15 12	11 8	7 4	3 0
0 0 1 1	func	rb	ra
4	4	4	4
	0000 (ADD)		
	0001 (SUB)		
	0100 (AND)		
	0101 (OR)		
	0110 (XOR)		
	1000 (SLT)		
	1001 (SLTU)		
	1010 (SGT)		
	1011 (SGTU)		
	1100 (SRL)		
	1101 (SLL)		
	1110 (SRA)		
	1111 (RSB)		

ADD přičte do registru *ra* hodnotu registru *rb*. SUB odečte od registru *ra* hodnotu registru *rb*. AND, OR a XOR provedou patřičnou funkci nad hodnotami registrů *ra* a *rb* a výsledek uloží do registru *ra*. Zbylé funkce se chovají obdobně jako v kapitole 2.3.4, jen místo hodnoty *imm* používají hodnotu registru *rb*.

2.3.6 Nepodmíněná skoková instrukce

Tabulka 2.6: Struktura nepodmíněných skokových instrukcí

15			12	11	0
0	0	1	0	imm	
4				12	

Tento instrukční formát má pouze jednu instrukci, jež způsobí skok na relativní adresu v rozsahu $2 \cdot imm$ vzhledem k adrese této instrukce.

2.3.7 Instrukce pro manipulaci s daty

Tabulka 2.7: Struktura instrukcí pro manipulaci s daty

15			12	11	8	7	4	3	0
0	0	0	1	func		rb		ra	
4				4		4		4	
0000 (EXTB)									
0001 (EXTW)									
0010 (EXTD)									
0100 (EXTBU)									
0101 (EXTWU)									
0110 (EXTDU)									
1000 (INSB)									
1001 (INSW)									
1010 (INSD)									
1100 (MSKB)									
1101 (MSKW)									
1110 (MSKD)									

Instrukční formát obsahuje instrukce pro manipulaci s datovými bloky menších velikostí než 64 bitů. EXTB extrahuje byte z registru *ra*, rozšíří

jeho znaménkový bit a výsledek uloží do registru *ra*. Hodnota registru *rb* je pak interpretována jako přirozeně zarovnaná adresa na byte a její tři nejnižší bity udávají index extrahovaného bytu. EXTBU pak nerozšiřuje znaménkový bit bytu. INSB vloží nejnižší byte registru *ra* do registru *ra* na index bytu daný hodnotou *rb* spočítanou stejným způsobem. Zbylé byty obsahují hodnotu nula. MSKB vynuluje byte registru *ra* s tímto indexem a výsledek uloží do registru *ra*. Zbylé funkce pak provádí obdobné operace pro slovo (dva byty) a dvojité slovo (čtyři byty).

2.3.8 Instrukce pro řídící registry

Tabulka 2.8: Struktura instrukcí pro řídící registry

15					11	10	9	7	6	4	3	0
0	0	0	0	1		func	x		cr		ra	
		5				1	3		3		4	
						0 (CRR)			pouze čtení		pouze zápis	
						1 (CRW)			pouze zápis		pouze čtení	

CRR přečte hodnotu řídícího registru *cr* a uloží ji do registru *ra*. CRW přečte hodnotu *ra* a uloží ji do řídícího registru *cr*. Pro účely těchto instrukcí RISC63 definuje indexy řídících registrů:

- `k0` – 000
- `k1` – 001
- `status` – 010
- `ivec` – 011
- `spc` – 100

2.3.9 Instrukce datového přesunu

Tabulka 2.9: Struktura instrukcí datového přesunu

15					10	9	8	7		4	3		0
0	0	0	0	0	1	x		rb			ra		
6						2		4			4		
pouze zápis													

Jediná instrukce v tomto instrukčním formátu přečte hodnotu registru *rb* a uloží ji do registru *ra*.

2.3.10 Instrukce bez operandů

Tabulka 2.10: Struktura instrukcí bez operandů

15					10	9	8	7		0
0	0	0	0	0	0	func		x		
6						2		8		
00 (NOP)										
01 (IRET)										

NOP zvyšuje pouze čítač instrukcí. IRET provede návrat z obsluhy přerušení (viz kapitola 2.2.2).

3 Návrh mikroarchitektury

Tato kapitola popisuje výslednou implementaci představené RISC63 architektury, která je implementována pomocí pětistupňového pipeliningu s jednotným hodinovým signálem v jazyce VHDL 93 [6]. Podporuje okamžitá přerušení a definuje jednoduché rozhraní pro instrukční a datovou paměť, což přináší široké spektrum aplikací. Procesor jako takový je pak připraven i pro nasazení do hradlového pole.[1]

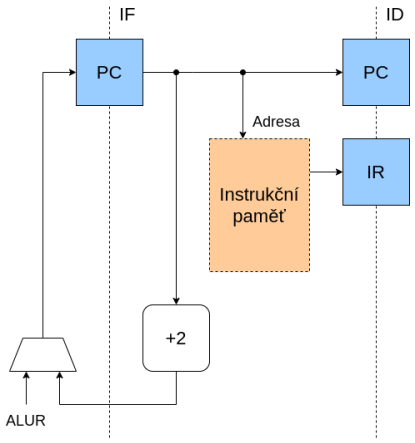
3.1 Popis pipeline

Jelikož celá mikroarchitektura stojí na principu zřetězeného zpracování instrukcí[2], je potřeba definovat stupeň, resp. fázi pipeline. Fázi pipeline chápeme jako množinu vstupních registrů a kombinační logiku bezprostředně závislou na jejich hodnotách. Výstup kombinační logiky je naveden do vstupních registrů následující fáze. Tím vzniká zřetězené zpracování instrukcí.

V rámci popisu pipeline jsou použita zjednodušená schémata jednotlivých fází, pomocí kterých si lze uvedené mechanismy lépe představit. V příloze B je pak zobrazeno schéma pipeline celého procesoru jako kompletní celek.

1. Původně se předpokládala demonstrace možností RISC63 na vývojové desce Digilent Basys 2. Ta ovšem svou kapacitou nedostačuje nárokům procesoru.
2. Tímto termínem se často v češtině pipelining označuje.

3.1.1 Fáze vyzvednutí instrukce (IF)



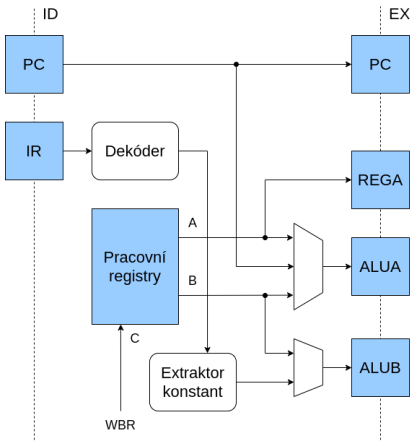
Obrázek 3.1: Struktura fáze vyzvednutí instrukcí

IF fáze je prvním stupněm pipeline, kterým instrukce projde při svém zpracování uvnitř procesoru. IF fáze nastaví hodnotu adresy další instrukce do registru PC a následně adresuje instrukční paměť. Pokud nebyl indikován žádný typ skoku řadičem procesoru, pak následující instrukce bude vyzvednuta z adresy, jejíž hodnota je o dva vyšší. Instrukce totiž mají 16 bitů a očekávají paměť s adresací po bytech. Pokud byl indikován skok, bude použit výsledek ALU jako další hodnota PC, protože právě tam byla spočítána adresa další instrukce.

Obrázek 3.1 však vzhledem ke své jednoduchosti nepostihuje přerušení. Pokud by došlo k přerušení, do PC registru se zapíše hodnota `ivec`. V případe návratu z přeruše se použije hodnota `spc`.

Ačkoli registr PC má architekturální velikost 64 bitů, skutečná velikost PC je 63 bitů, protože nejnižší bit každé adresy instrukce je nula. V důsledku se pak nepřičítá hodnota dva, ale jedna.

3.1.2 Fáze dekódování instrukce (ID)



Obrázek 3.2: Struktura fáze dekódování instrukcí

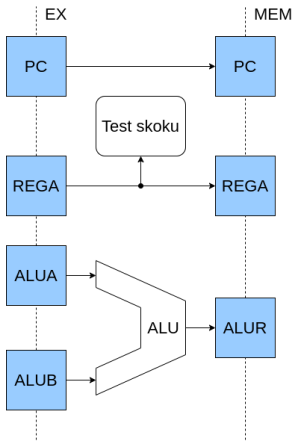
Po vyzvednutí instrukce z paměti je uložena v registru IR[3], odkud probíhá její další zpracování. Jsou z ní extrahovány indexy pracovních registrů, které jsou následovně přečteny. Vyšší byte instrukce je poslán do dekodéru, aby se určil formát instrukce a nastavily řídicí signály jako např. operační kód pro ALU. Na základě řídicích signálů extraktor konstant vybere zamýšlený rozsah bitů z instrukce jako konstantu a znaménkově bity rozšíří na 64 bitů. Taktéž multiplexory obdrží řídicí signály a propagují dál požadované operandy pro ALU.

Dekódování instrukcí RISC63 architektury se ukazuje být docela komplikované. Vzhledem k typu ISA registr-registr a nejednotnosti konstant v instrukčním slově existuje mnoho výjimek, na které se musí brát zřetel v dekodéru. To vede ke zvýšení jeho složitosti.

V rámci této fáze probíhá i zápis do pracovních registrů z fáze WB jako výsledek některé z předcházejících instrukcí. Tento výsledek je okamžitě dostupný pro aktuálně dekódovanou instrukci. Mezi ID a WB je více fází pipeline, a tak výsledky libovolné předcházející instrukce dostupné být nemusí. Snaha použít nějaký takový výsledek vede k datovému hazardu (viz kapitola 3.1.6).

3. Instruction Register – instrukční registr

3.1.3 Fáze provádění instrukce (EX)

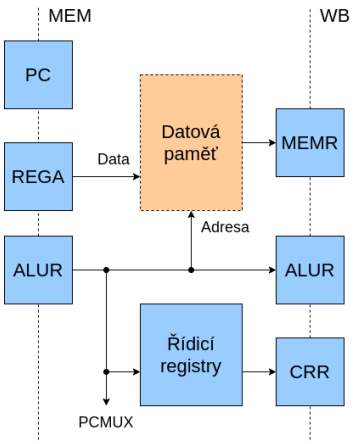


Obrázek 3.3: Struktura fáze provádění instrukcí

EX fáze provádí samotný výpočet požadovaný aktuální instrukcí v ALU. Vstupem jsou jí dva operandy připravené z ID fáze. EX fáze zároveň určuje, zda se podmíněné skoky vykonají, nebo ne. K tomuto účelu byla doručena hodnota registru A ve speciálním registru, který je nezávislý na zvolených operandech ALU. Při skocích se totiž v ALU počítají výsledné adresy skoků a otestovat, zda se podmíněný skok provede, se musí jinde.

ALU jednotka implementuje v sobě dva moduly z důvodu dekompozice návrhu. Prvním modulem je 64bitová univerzální sčítačka. Na tuto jedinou sčítačku jsou mapovány všechny instrukce součtu, rozdílu i aritmetického porovnávání. Druhý modul umožňuje manipulaci s datovými bloky menšími než 64 bitů. Provádí extrakce, vkládání a maskování.

3.1.4 Fáze přístupu do paměti (MEM)



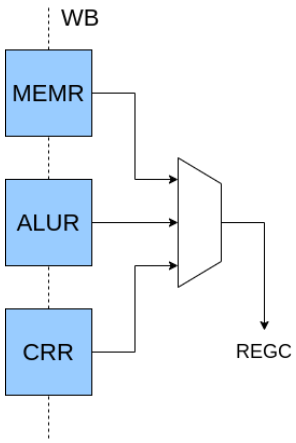
Obrázek 3.4: Struktura fáze přístupu do paměti

V MEM fázi může procesor komunikovat s datovou pamětí. Pro zápis používá registr A a jako adresu výsledek ALU. Obdobně jako pro cílové adresy skoků se v ALU počítají i cílové adresy dat. Z paměti lze číst výsledky všech předcházejících instrukcí bez rizika hazardu (viz kapitola 3.2). Mimoto se v této fázi přistupuje i k řídícím registrům a zápisy do nich mají také okamžitý efekt na následující instrukce.

MEM fáze také posílá signály související se skoky do řadiče procesoru. Pokud se jedná o obyčejné instrukce skoku, pak v registru výsledku ALU je již připravena adresa další instrukce. Pakliže jde o instrukci návratu z přerušení, pochopitelně se použije hodnota `spc` registru, která je přímo vyvedena z řídících registrů.

Vzhledem k tomu, že se jedná o první fázi ovlivňující architekturální stav datového úložiště procesoru, má velmi důležitou roli i pro přerušení. Pokud je totiž přerušení přijato, jsou veškeré zápisy této fáze zrušeny a PC hodnota registru této fáze je uložena do `spc`, odkud se pak bude pokračovat, protože žádný výsledek se nezapsal. Adresa další instrukce je hodnota registru `ivec`, která je taktéž vyvedená z řídících registrů. Vliv přerušení a skoků na pipeline je popsán v kapitole 3.1.6.

3.1.5 Fáze zápisu výsledku (WB)



Obrázek 3.5: Struktura fáze zápisu výsledku

Fáze WB je poslední fází zpracování instrukce. Zapisuje výsledek do pracovního registru procesoru. Může to být přečtená hodnota z paměti, ALU výsledek, nebo hodnota řídícího registru.

3.1.6 Pipeline hazardy

Zpracování instrukcí zřetězeným způsobem bývá doprovázeno negativním jevem označovaným jako hazard. Jedná se o stav procesoru, který může bezprostředně vést k chování procesoru, které neodpovídá sekvenční povaze vykonávaných instrukcí. Pro korektní chování procesoru za každých okolností se musí všechny možné hazardy identifikovat a ošetřit na úrovni hardware. Pokud mikroarchitektura trpí nějakým hazardem, lze k tomu přizpůsobit i software. Ten se pak musí vyhýbat kritickým situacím, které k iniciaci hazardu vedou. Nejedná se však o ideální řešení. Pro kontext této práce jsou důležité pouze dva typy hazardů – řídící hazard a datový hazard.

Řídící hazard je úzce spojený se skoky a přerušením. V momentě, kdy se provádí v navrženém procesoru skok, vstup do rutiny přerušení nebo návrat z ní, řadič procesoru automaticky nastaví hodnotu reset signálu ID, EX a MEM fází jako aktivní na dobu jednoho hodinového signálu. To způsobí, že žádný výsledek jakékoliv již vyzvednuté

instrukce po instrukci skoku nebude uložen. Mezitím IF fáze už vyzvedává instrukci z nové adresy PC registru a WB fáze dokončuje zápis instrukce před skokem. Oba tyto kontexty se pak bezkonfliktně potkají v ID fázi. Řídící hazardy jsou tedy ošetřené.

Datové hazardy momentálně ošetřeny nejsou, je potřeba se jim vyhnout softwarově. Pro vznik datového hazardů existuje jednoduché pravidlo. Pokud instrukce čte hodnotu registru, do kterého nějaká ze tří předcházejících instrukcí zapisovala, obdrží neaktuální hodnotu. A to je datový hazard. Je způsoben skutečností, že se pracovní registry čtou ve fázi ID, kdežto výsledek se do nich zapisuje až ve fázi WB.

K řešení takových datových hazardů se obvykle používají dva přístupy. Buďto se zastaví část pipeline a počká se, až bude výsledek dostupný, nebo se výsledek propaguje zpátky z několika fází, nejen z poslední. Čekání na výsledek při čtení paměti se ovšem reálně vyhnout nedá ani v tomto případě.

3.2 Rozhraní pro komunikaci s pamětí

Procesor pro svůj korektní běh potřebuje instrukční a datovou paměť výhradně s asynchronním čtením. Datová paměť navíc musí podporovat synchronní zápis za použití stejného hodinového signálu jako procesor. Výsledná mikroarchitektura nepodporuje sdílení paměti, resp. zamítnutí požadavku přístupu k paměti, tudíž by mohly nastat problémy v případě potřeby přetvořit výslednou paměťovou hierarchii směřující k jedné společné operační paměti pro instrukce i data, čemuž architektura RISC63 jako taková nijak nebrání.

3.2.1 Rozhraní pro instrukční paměť

Procesor komunikuje s pamětí instrukcí za použití uvedených portů:

1. Výstup 63bitový `o_imem_addr`[4]
2. Vstup 16bitový `i_imem_rd_data`

První uvedený port reprezentuje adresu a druhý přečtenou instrukci z této adresy. Čtení probíhá neustále.

4. Pohybujeme se v 64bitovém adresním prostoru bytů a tento port adresuje paměť po slovech. Nejnižší bit tedy není nikdy potřeba.

3.2.2 Rozhraní pro datovou paměť

Komunikace s datovou pamětí používá následující porty:

1. Výstup jednobitový `o_dmem_we`
2. Výstup 61bitový `o_dmem_addr`[5]
3. Výstup 64bitový `o_dmem_wr_data`
4. Výstup 64bitový `i_dmem_rd_data`

První uvedený port povoluje zápis do paměti. Druhý reprezentuje adresu paměti. Zbylé dva porty reprezentují zapisovaná a přečtená data. Čtení paměti probíhá neustále.

3.3 Úvod do VHDL projektu

Vytvořený VHDL projekt, který je elektronickou přílohou této práce, dodržuje určité konvence a adresářovou strukturu. V této kapitole je podstatná část této problematiky vyjasněna.

3.3.1 Použité konvence

Každý datový objekt použitý ve zdrojových kódech projektu má ve svém identifikátoru předponu. Mimo datové objekty jsou předpony použity i pro identifikátory nějakých dalších konstrukcí VHDL. Tento přístup byl zaveden, protože na mnohá místa ve VHDL kódu lze použít více datových objektů. Následuje výčet použitých předpon:

- `i_` – vstupní porty
- `o_` – výstupní porty
- `s_` – signály
- `v_` – proměnné
- `c_` – konstanty

5. Bitová šířka je zvolena obdobně jako u `o_imem_addr` portu uvedeného dříve.

- `t_` – vlastní datové typy
- `f_` – funkce

Uvedené předpony se neskládají a zanikají zároveň s asociovaným datovým objektem. Např. výstupní port `o_cr_ie` mapujeme na signál `s_cr_ie`, nikoli na `s_o_cr_ie`. Pokud to jednoznačně nelze, zachováme i předponu. Snažíme se tomu však předejít.

Další adaptovanou konvencí v projektu je po této předponě uvést také předponu reprezentující komponentu, které této signál náleží (pokud nějaká existuje). Nejpoužívanějšími předponami jsou:

- `cu_` – řadič procesoru
- `cr_` – řídící registry
- `if_` – IF fáze
- `id_` – ID fáze
- `ex_` – EX fáze
- `mem_` – MEM fáze
- `wb_` – WB fáze

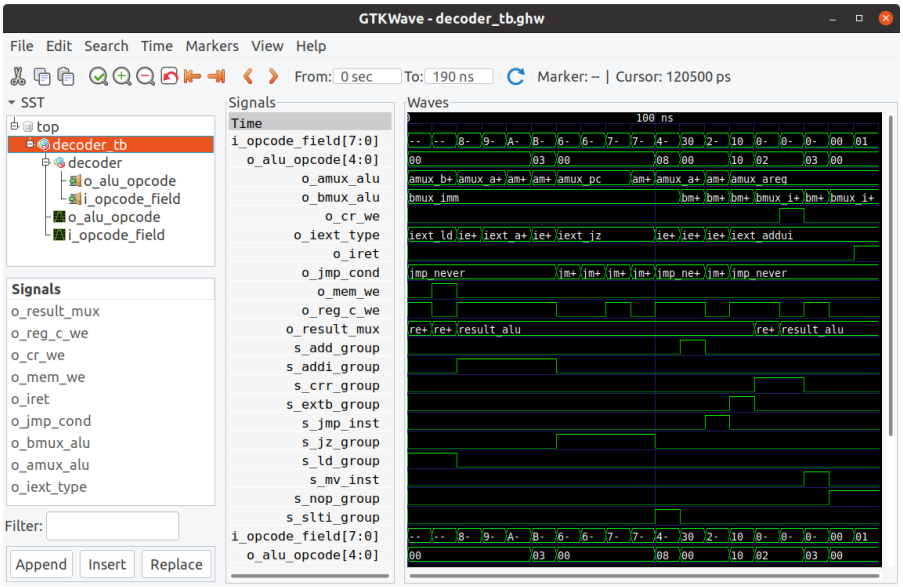
3.3.2 Adresářová struktura

Projekt VHDL má následující adresářovou strukturu:

```
risc63
├── rtl ..... VHDL popis mikroarchitektury
│   ├── if ..... vyzvednutí instrukce
│   ├── id ..... dekódování instrukce
│   ├── ex ..... provádění instrukce
│   ├── mem ..... přístup do paměti
│   └── wb ..... zápis výsledku
├── test ..... test bench moduly
└── build ..... build systém, makefile
```

4 Simulace mikroarchitektury

Simulace mikroarchitektury je hlavním měřítkem korektnosti modelu navrženého procesoru ve VHDL. K simulaci se používá program GHDL, který lze nainstalovat pomocí dostupných připravených binárních distribucí.[1] Případně jej lze zkompilovat pro konkrétní systém podle návodu uvedeného v dokumentaci.[2] Pro potřeby graficky zobrazit průběh přechodů signálů zvolené komponenty je používán program GTKWave[3]. Viz obrázek 4.1.



Obrázek 4.1: Ukázka programu GTKWave

Samotná simulace a testování jsou pak prováděny prostřednictvím test bench souborů popsaných též v jazyce VHDL.[4] Test bench soubory provádí tzv. black-box testování. Vybere se VHDL komponenta a obsáhne se v test bench modulu. Poté se řídí její vstupy a v závislosti na tom se kontrolují její výstupy vůči očekávaným výstupům.

1. `https://github.com/ghdl/ghdl/releases`
2. `https://ghdl.readthedocs.io/en/latest/getting/index.html`
3. `http://gtkwave.sourceforge.net`
4. Určitá podmnožina jazyka VHDL je použitelná pouze pro simulační účely.

4.1 GHDL simulátor

GHDL může být zkompilován tak, aby používal jeden z několika podporovaných back endů. V závislosti na back endu pak mohou existovat malé rozdíly v obsluze GHDL, avšak hlavní funkcionalita je se všemi back endy ekvivalentně dostupná.

Pro manuální spuštění testu pomocí GHDL je potřeba nejprve importovat všechny dotčené moduly (zahrnuje závislosti) do pracovní knihovny. Lze toho dosáhnout příkazem `ghdl -i <soubory>`. Poté je potřeba zkompilovat entitu testu. Toho lze dosáhnout příkazem `ghdl -m <entita>`. Tento příkaz zkompiluje i entity, které zadaná entita obsahuje. Pro samotné spuštění testu se pak používá příkaz `ghdl -r <entita>`.

4.2 Vlastní build systém

V průběhu vývoje RISC63 procesoru se ukázalo, že standardní API[5] programu GHDL nebude dostačovat pro další postup. Zejména bylo třeba automaticky kompilovat všechny entity a spustit všechny testy. Také byly požadovány možnosti zkompilovat jen jednu entitu a totéž se spuštěním testu. U spouštění testu pak volitelně spustit i náhled jeho průběhu v programu GTKWave. Každá taková funkce musela být dostupná prostřednictvím jednoho příkazu.

Jako řešení bylo zvoleno vytvoření POSIX[6] `makefile` [7], který byl umístěn do adresáře `build`, kde také generuje pomocné soubory při vykonávání požadovaných úkonů. Tento soubor definuje několik cílů, které lze sestavit prostřednictvím `make` příkazu vyvolaného z adresáře `build`. Následuje seznam nejdůležitějších:

- `elab` – zkompiluje všechny entity
- `test` – spustí testy všech test bench entit
- `elab_a` – zkompiluje entitu, jejíž název je uložen v makru `a`
- `test_a` – spustí test zadané entity

5. Application Programming Interface – rozhraní pro programování aplikací
6. Portable Operating System Interface – přenosné rozhraní pro operační systémy

- `view_a` – spustí test zadané entity a zobrazí jeho průběh
- `clean` – smaže všechny vygenerované soubory

V případě, že je do projektu přidán nový VHDL soubor, musí se aktualizovat i interní seznam projektových VHDL souborů uvedený v `makefile`. Celý build systém byl otestován v Linuxovém prostředí (Ubuntu 18.04.4).

Závěr

Výsledkem práce je architektura vlastní instrukční sady, mikroarchitektura, která z ní vychází, a popis mikroarchitektury v jazyce VHDL včetně testů dokládajících její funkčnost. V průběhu práce byl stručně popsán celý proces vývoje modelu procesoru od počáteční definici pojmů přes volbu vlastností instrukční sady až po detailní popis práce pipeline cílené mikroarchitektury. Mnohá rozhodnutí jsou doložena odkazem na již existující procesorové architektury, protože mají klíčový vliv na tuto práci.

Volba instrukčního slova 16 bitů vedla k omezením při návrhu instrukční sady i při následné implementaci v mnohých ohledech. Několikrát bylo na tuto skutečnost poukázáno jako na negativní aspekt menších instrukcí. Práce tak naplnila původní motivaci prozkoumat dopady volby 16bitových instrukcí na návrh modelu procesoru a jeho architektury.

I navzdory obtížím při snaze implementovat mikroarchitekturu efektivně, se podařilo navrhnout a zrealizovat přerušovací systém procesoru. Způsob fungovaní přerušení je velmi jednoduchý a přitom důmyslný. Přerušení jsou navíc akceptována okamžitě, což nebývá pravidlem při použití pipeline.

Klíčová vlastnost, která procesoru chybí, je schopnost se vypořádat s datovými hazardy v pipeline. V práci byly zmíněny dvě možné metody řešení tohoto problému a lze to považovat za směr budoucího vývoje RISC63 procesoru. Eliminací datových hazardů se totiž sníží velikost aktuálních programů a RISC63 bude moci dostát svému původnímu záměru.

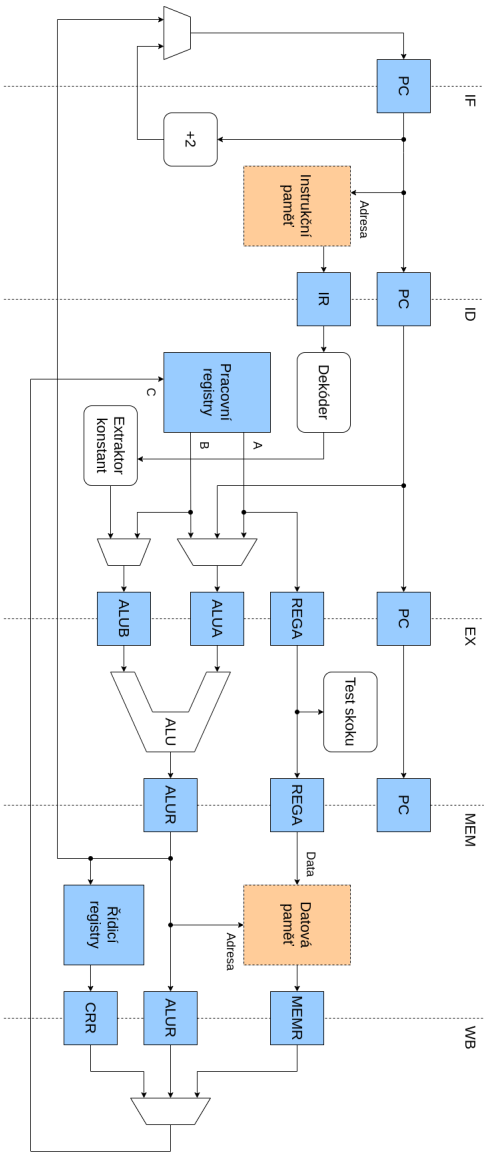
Bibliografie

1. BUNDA, J.; FUSSELL, D.; ATHAS, W.C.; JENEVEIN, Roy. 16-bit Vs. 32-bit Instructions For Pipelined Microprocessors. In: 1993, s. 237–246. ISBN 0-8186-3810-9. Dostupné z DOI: 10.1109/ISCA.1993.698564.
2. BURGER, D. et al. Scaling to the end of silicon with EDGE architectures. *Computer*. 2004, roč. 37, č. 7, s. 44–55.
3. WATERMAN, A.; ASANOVIĆ, K. (ed.). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.
4. WATERMAN, A.; LEE, Y.; PATTERSON, D.; ASANOVIĆ, K. *RISC-V Compressed Extension* [online]. 2015 [cit. 2020-07-15]. Dostupné z: https://riscv.org/wp-content/uploads/2015/06/riscv-compressed-workshop-june2015.pdf.
5. ALPHA ARCHITECTURE COMMITTEE et al. *Alpha Architecture Reference Manual*. Ed. SITES, R. L. Digital Press, 2014.
6. IEEE Standard VHDL Language Reference Manual. *ANSI/IEEE Std 1076-1993*. 1994, s. 1–288. Dostupné z DOI: 10.1109/IEEESTD.1994.121433.
7. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. 2018, s. 1–3951. Dostupné z DOI: 10.1109/IEEESTD.2018.8277153.

A Elektronické přílohy

Jedinou elektronickou přílohou práce je implementace popsané mikroarchitektury. Krátký úvod k implementaci je uveden v kapitole 3.3.

B Schéma pipeline



Obrázek B.1: Struktura pipeline implementované mikroarchitektury