VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE GENERICKÉHO PROCESORU V FPGA

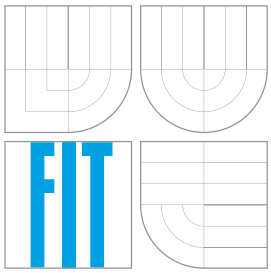
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MIKUŠEK

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE GENERICKÉHO PROCESORU V FPGA

IMPLEMENTATION OF GENERIC PROCESSOR IN FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MIKUŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ MARTÍNEK

BRNO 2007

Zadání

1. Seznamte se s technologií programovatelných hradlových polí FPGA a dostupnými nástroji pro syntézu a implementaci obvodů do FPGA hradlových polí.
2. Seznamte se se současnými architekturami vestavěných procesorů a metodami jejich návrhu.
3. Navrhněte generický procesor, který bude nastavitelný sadou parametrů. Mezi tyto parametry uvažujte například šířku datové cesty nebo velikosti paměťových prvků.
4. Proveďte implementaci navrženého řešení v jazyce VHDL nebo HandleC a jeho funkci ověřte simulací.
5. Realizujte funkční prototyp procesoru a ověřte jeho správnost na kartě COMBO6.
6. V závěru diskutujte vlastnosti vytvořené implementace a možnosti dalšího pokračování projektu.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Tato práce se zabývá studií architektur vhodných pro vestavěné procesory, mezi něž patří i přenosem spouštěné architektury (TTA). Tyto architektury se programují uvedením přenosů dat a operace se spouští jako jejich vedlejší efekt. V tradičních operací spouštěných architekturách (OTA) program přímo udává požadované operace. Přesuny dat jsou v režii hardware a nemohou být řízeny a optimalizovány kompilátorem v době kompilace. Tento přístup přináší spoustu výhod po stránkách hardwarových i softwarových. Cílem této práce bylo provést návrh a implementaci ukázkového TTA procesoru v jazyce VHDL s následným ověřením realizace v hradlovém poli FPGA. Tento procesor je navržen do značné míry jako generický, tj. nastavitelný sadou parametrů, jako je datová šířka, počty sběrnic, atd.

Klíčová slova

přenosem spouštěné architektury, VLIW, architektury procesorů, VHDL, COMBO6X, FPGA, Virtex-II Pro

Abstract

This thesis studies processor architectures suitable for embedded processors. This includes Transport Triggered Architectures (TTA). TTA is programmed by specifying data transport; operations are triggered as a side effect of data transports. In traditional Operation Triggered Architectures (OTA) requested operations are determined by program. Data transports are handled internally by hardware so it's impossible to control and optimize data transfer by compiler. This approach brings an advantage of hardware and software aspects. The aim of this thesis is to design and implement a sample TTA processor in VHDL followed by realization in FPGA. This processor is designed in a generic manner, i.e. customized by set of generic parameters such as data width, number of buses, etc.

Keywords

transport triggered architectures, VLIW, processor architectures, VHDL, COMBO6X, FPGA, Virtex-II Pro

Citace

Petr Mikušek: Implementace generického procesoru v FPGA, diplomová práce, Brno, FIT VUT v Brně, 2007

Implementace generického procesoru v FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Tomáše Martínka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Mikušek
22. května 2007

Poděkování

Především bych rád poděkoval vedoucímu své diplomové práce panu Ing. Tomáši Martínkovi za odborné vedení a čas věnovaný konzultacím této práce. Také bych chtěl poděkovat kolegům z projektu Liberouter za zajištění technické podpory při návrhu a implementaci.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Procesory hrají důležitou roli ve všech oblastech lidského života. Každý rok se vyrobí miliardy procesorů, které se uplatní v mnoha druzích zařízení, počínaje od vysoce jednoúčelových systémů až po osobní počítače pro obecné použití. Je jasné, že tyto systémy musí splňovat velice rozdílné požadavky. Jsou totiž navrženy pro úplně odlišné použití s různými požadavky na cenu a výkon.

Návrhář počítačového systému je postaven před skoro neřešitelný problém, když má navrhnout a implementovat architekturu systému založeném na procesoru. Při návrhu musí být do úvahy vzata četná rozhodnutí, která ovlivňují konečný výsledek a těžko se dopředu odhaduje jejich dopad. Řešením tohoto problému může být použití automatizovaného návrhu, který postihne celou škálu aplikací. Potřebujeme takovou šablonu architektury, která by splňovala požadavky na škálovatelnost, flexibilitu, programovatelnost, modularitu a softwarovou kompatibilitu. Škálovatelnost a flexibilita je potřeba pro pokrytí širokého okruhu řešení, programovatelnost je potřeba pro možnost psát programy ve vyšším programovacím jazyce, modularita je nezbytná pro účinnou a automatickou syntézu výsledného systému a softwarová kompatibilita nám zaručuje použitelnost stávajícího kódu.

Existující architektury nemohou splnit všechny tyto požadavky zároveň. Nejslibnějším kandidátem jsou architektury s velmi dlouhým instrukčním slovem (VLIW, *Very Long Instruction Word*). Architektury VLIW poskytují podporu pro využívání instrukčního paralelismu (ILP, *Instruction Level Paralelism*) použitím velkého množství funkčních jednotek. Bohužel ale tato architektura nesplňuje všechny požadavky na škálovatelnost, flexibilitu a modularitu. V práci bude dále ukázáno, že realizace těchto architektur je zbytečně složitá. Proto byla navržena nová třída architektur, která se nazývá *přenosem spouštěné architektury* (TTA, *Transport Triggered Architectures*). [1]

Na TTA architektury může být pohlíženo jako na sadu funkčních jednotek a souborů registrů propojených pomocí propojovací sítě. Operace funkčních jednotek se spouští jako vedlejší efekt přenosů dat. V tradičních operací spouštěných architekturách (OTA, *Operation Triggered Architectures*) program přímo udává požadované operace. Přesuny dat, které tyto operace potřebují, se dějí v režii hardware a nemohou být řízeny a optimalizovány kompilátorem v době kompilace. Tento přístup přináší spoustu výhod po stránkách hardwarových i softwarových.

Cílem této práce je nastudovat TTA architektury a provést návrh a implementaci ukázkového TTA procesoru v jazyce VHDL s následným ověřením realizace v hradlovém poli FPGA. Tento procesor by měl být do značné míry generický, tj. nastavitelný sadou parametrů, jako je datová šířka, počty sběrnic, atd.

Struktura této práce je následující. Kapitola 2 krátce diskutuje současné architektury

vestavěných procesorů. Hlavní obsahem této kapitoly je popis hardwarových a softwarových aspektů přenosem spouštěných architektur. Kapitola 3 se zaměřuje na návrh a implementaci generických komponent TTA architektury. Mezi tyto komponenty patří jednotka načítání instrukcí, jednotka dekódování instrukcí, predikační jednotka, propojovací síť, ale i celá řada funkčních jednotek a souborů registrů. Kapitola 4 ukazuje realizaci ukázkového TTA procesoru sestavením komponent prezentovaných v předcházející kapitole. Funkčnost navrženého procesoru je ověřena simulací a jeho funkční prototyp realizován na kartě COMBO6X. Poslední závěrečná kapitola 5 diskutuje vlastnosti vytvořené implementace a jejich zhodnocení.

V porovnáním se semestrálním projektem byla po důkladném nastudování TTA architektur rozšířena kapitola popisující TTA architektury a proveden detailní návrh a implementace generických jednotek. Nově byla přidána kapitola s realizací a konkrétní implementací ukázkového TTA procesoru.

Kapitola 2

Přenosem spouštěné architektury

Možnost překrytí zahájení a provedení více instrukcí v rámci jednoho procesoru se nazývá instrukční paralelismus (ILP, *Instruction Level Parallelism*). Využívání instrukčního paralelismu se stalo důležitou metodou zvyšování výkonu nejnovějších mikroprocesorů.

V superskalárních procesorech (např. Intel Pentium [9]) hardware procesoru za běhu detekuje a řeší závislosti mezi operacemi instrukčním proudu. Tento přístup poskytuje binární kompabilitu s předchozími generacemi architektur, ale trpí složitostí hardwaru a dlouhou dobou návrhu, což dělá superskalární architektury nezajímavé z pohledu využití ve vestavěných systémech.

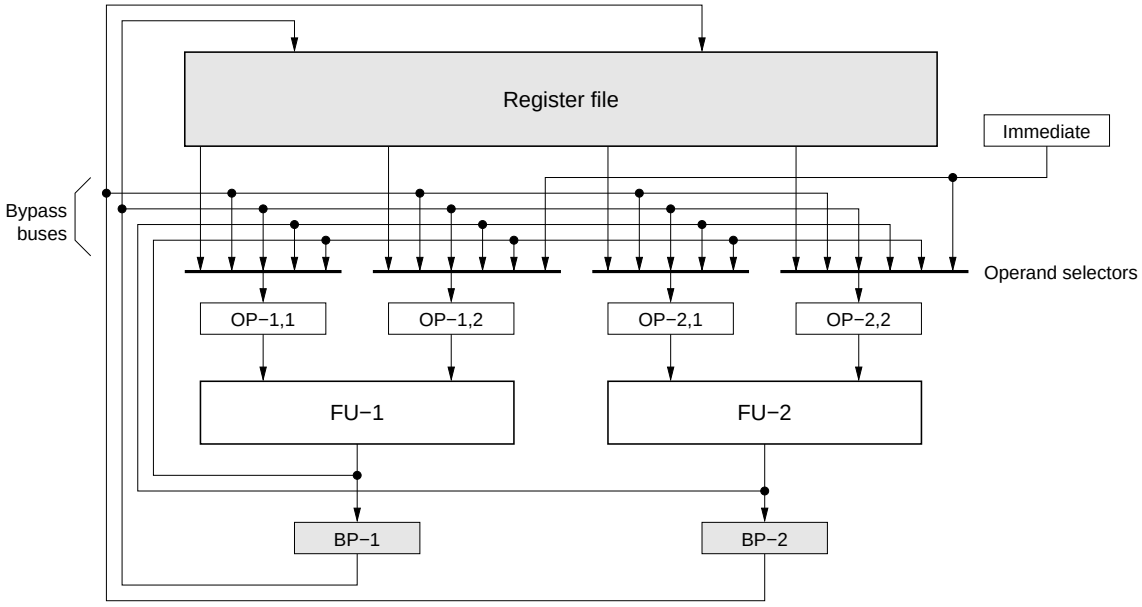
Procesory VLIW (*Very Long Instruction Word*) vydávají pouze jednu instrukci za takt. Instrukce sestává z několika RISCových operací, jako je sčítání, nahrání dat z paměti nebo skok. Datové závislosti jsou ale řešeny v době kompilace pomocí sofistikovaného kompilátoru provádějící plánování instrukcí. Kompilátor pro procesory VLIW má velice dobrou kontrolu nad zdroji procesoru a hraje důležitou roli ve využívání zdrojů a zvyšování výkonnosti. K dosažení tohoto úkolu musí kompilátor znát detailní model cílového procesoru a udržovat si přehled o stavu procesoru a využití zdrojů. Díky tomu jsou procesory VLIW jednodušší, flexibilnější a poskytují dobrou škálovatelnost. Mezi komerčně význačné VLIW procesory patří rodina DSP procesorů TMS320C6xxx [10] od společnosti Texas Instruments nebo rodina multimediálních procesorů TriMedia [11] společnosti NXP Semiconductors (dříve Philips Semiconductors).

Navzdory dobrým vlastnostem architektur VLIW se může jejich datová cesta stát příliš složitou zvláště tehdy, když je škálována pro vysoký výkon. Princip co největšího zjednodušení hardware procesoru a přenechání úsilí kompilátoru v době překladu je rozpracován ještě dále v třídách architektur procesorů nazývaných *přenosem spouštěné architektury* (TTA, *Transport Triggered Architectures*) [1].

V kapitole 2.1 si povíme o odvození TTA architektury z architektury VLIW. Kapitola 2.2 pojednává o hardwarových aspektech TTA architektur, o softwarových aspektech pojednává kapitola 2.3. Konečně kapitola 2.4 ukáže realizaci konkrétního TTA procesoru.

2.1 Od architektury VLIW k TTA

Datová cesta procesoru VLIW se skládá z funkčních jednotek (FU, *functional unit*), které jsou připojeny přes předávací síť (*bypassing network*) k souboru registrů (RF, *register file*), jak je ilustrováno na obrázku 2.1. Předávací síť je potřebná k předávání výsledku z výstupu funkční jednotky do vstupního registru jiné jednotky v případě datového konfliktu čtení



Obrázek 2.1: Datová cesta VLIW procesoru se dvěma funkčními jednotkami

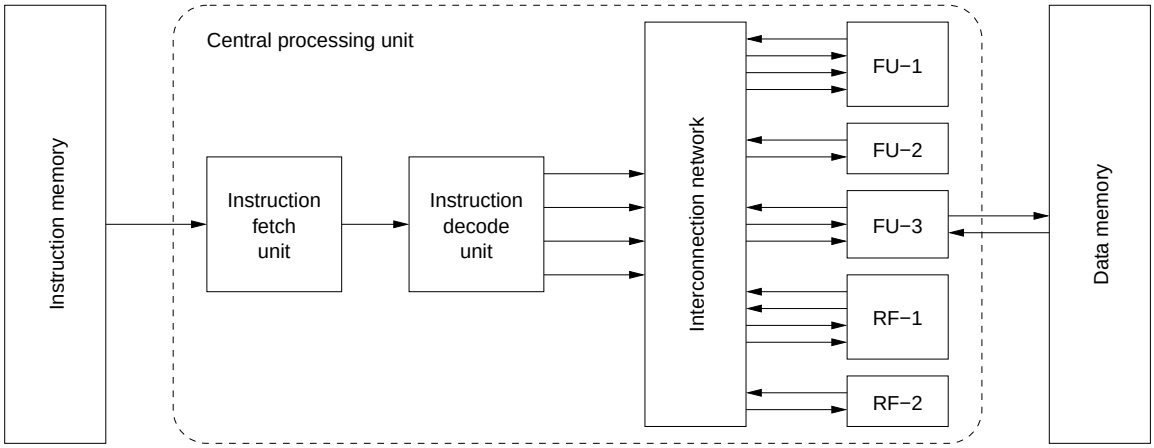
po zápisu (RAW, *Read After Write*). Předávání hodnot z výstupů všech FU na vstupy všech FU vyžaduje křížový přepínač, jehož složitost se zvyšuje kvadraticky se vzrůstajícím počtem funkčních jednotek. Navíc plná propustnost této propojovací sítě je zřídka využita, ani když jsou všechny jednotky zaneprádněné. [1]

Složitost souboru registrů se též může stát úzkým hrdlem architektury VLIW. Pro každou funkční jednotku jsou potřeba dva čtecí a jeden zápisový port[1]. To je pouze pro ten nejhorší případ, kdy každá funkční jednotka potřebuje provádět dvě čtení a jeden zápis do souboru registrů. Ve většině VLIW procesorů jsou k jednomu souboru registrů připojeny čtyři nebo méně funkčních jednotek. Při velmi velkém počtu portů bohužel klesá výkonnost.

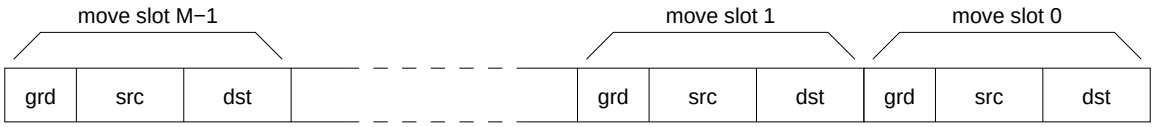
Přenosem spouštěné architektury byly vyvinuty, aby snížily složitost VLIW architektur přenecháním řízení provozu mezi registry na úroveň programu. Jinými slovy, přenosy dat se stávají viditelnými na úrovni architektury a mohou být řízeny a optimalizovány pomocí kompilátoru. Architektury TTA jsou organizovány jako sada funkčních jednotek a souborů registrů, které jsou propojeny pomocí propojovací sítě. Některé funkční jednotky mohou mít propojení s vnějškem procesoru, např. s externí pamětí. Tato organizace je ilustrována na obrázku 2.2.

Architektury TTA připomínají architekturu VLIW v tom, že umožňují provádět několik operací za takt. Hlavní rozdíl je v tom, jakým způsobem jsou operace programovány a prováděny. Instrukce u VLIW architektur specifikuje RISCové operace, zatímco instrukce u TTA architektur specifikuje přenosy dat. Operace jsou spouštěny jako vedlejší efekt těchto přenosů dat. Cíl přenosu implicitně udává, jaká operace se má nad daty provést.

[1]Za předpokladu sdíleného souboru registrů a funkčních jednotek se dvěma zdrojovým a jedním cílovým operandem.



Obrázek 2.2: Uspořádání TTA architektury



Obrázek 2.3: Obecný instrukční formát TTA procesoru

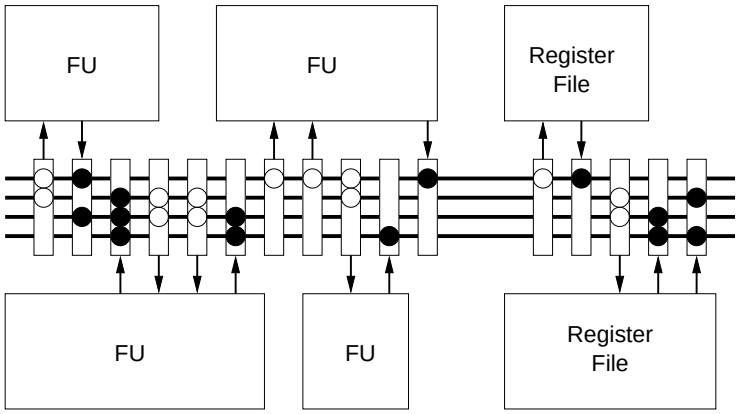
2.2 Hardwarové aspekty

Architektury TTA sestávají z omezeného počtu stavebních bloků. V podstatě jsou TTA procesory stavěny vhodným propojením funkčních jednotek a propojovací sítě. Na soubor registrů může být nahlíženo jako na speciální druh funkční jednotky. Funkční jednotky jsou zcela nezávislé navzájem i na propojovací síti. Funkční jednotky proto mohou být odděleně navrhovány, nezávisle řetězeny a mohou podporovat jakýkoliv typ operací. Modularita TTA architektur umožňuje zautomatizovat proces návrhu hardwaru. Různé TTA procesory mohou být snadno vytvářeny kombinací těchto stavebních bloků. [1]

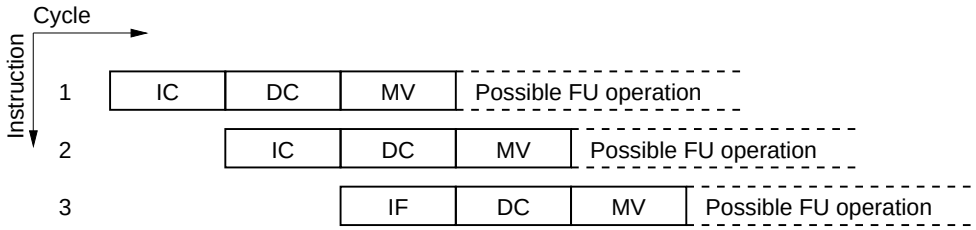
2.2.1 Propojovací síť

Propojovací síť umožňuje funkčním jednotkám a souborům registrů vyměňovat si data. Propojovací síť se skládá ze dvou jednoduchých komponent: sběrnic (*bus*) a zásuvek (*socket*). Sběrnice umožňují nejen přenos dat, ale také provádějí distribuci signálů, které řídí přenosy: zdrojové a cílové identifikátory registru a signál pro blokování procesoru.

TTA instrukce pro procesor s M sběrnicemi, zobrazena na obrázku 2.3, sestává typicky z M částí, která každá specifikuje nezávislý a současný přenos dat ze zdroje do cíle. Zdrojem se myslí výstup a cílem zase vstup funkční jednotky. Rozhraní mezi sběrnicemi a funkčními jednotkami poskytují zásuvkami, které implementují programovatelné propojení mezi funkčními jednotkami a sběrnicemi. Každá zásuvka je připojena k jedné nebo více sběrnicím a k jednomu nebo více registrům jedné funkční jednotky. Vstupní zásuvky jsou v podstatě vstupní multiplexory, které vybírají hodnotu na jedné se sběrnic a zapisují ji do cílového registru. Výstupní zásuvky jsou demultiplexory a nahrávají obsah zdrojového registru na jednu nebo více připojených sběrnic. Zdrojové a cílové položky jsou zasílány přes řídicí cestu sběrnic ke všem připojeným zásuvkám. Zásuvka připojená k požadovanému registru je aktivována číslem registru a propouští data v požadovaném směru.



Obrázek 2.4: Propojovací síť TTA procesoru



Obrázek 2.5: Schéma třístupňového přenosového zřetězení

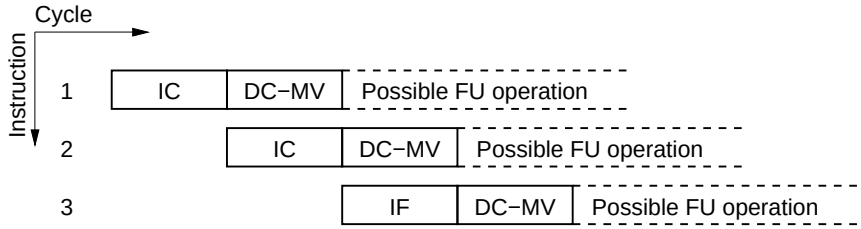
Propojovací síť může být plně propojená nebo částečně propojená. Plně propojená síť usnadňuje úlohu přiřazení přenosů na sběrnice. Plně propojená síť není příliš efektivní, protože zabírá větší plochu, pracovní frekvence je nižší a má větší příkon než částečně propojená síť. Pro aplikačně-specifické procesory by měla konektivita propojovací sítě odpovídat komunikačním požadavkům konkrétních aplikací a cenovým omezením. Jelikož jsou přenosy programovány explicitně, TTA kompilátor může účinně zajistit směrování datových přenosů přes částečně propojenou síť.

Částečně propojená propojovací síť složená ze čtyř sběrnic, propojující čtyři funkční jednotky a dva soubory registrů přes osm vstupních a osm výstupních zásuvek, je ilustrována na obrázku 2.4. Možné propojení je naznačeno prázdným kolečkem pro výstupní zásuvku a plným pro vstupní zásuvku.

2.2.2 Přenosové zřetězení

Vykonávání instrukcí může být efektivně zřetězeno na TTA architekturách. Toto zřetězení se nazývá přenosové zřetězení (*transport pipelining*). Zřetězení funkčních jednotek, popsané v kapitole 2.2.3, je taktéž podporováno a může být navrženo nezávisle na přenosovém zřetězení. Typicky jsou přenosy dat vykonávány v třístupňové zřetězené lince, která se skládá ze stupňů načtení instrukce (IF, *instruction fetch*), dekódování (DC, *instruction decode*) a přesunu (MV, *move*). Obrázek 2.5 ukazuje toto schéma zřetězení z pohledu instrukčního toku. Stupně dekódování a přesunu mohou být sloučeny v jeden společný stupeň, jelikož dekódování instrukcí u TTA architektur je velice triviální. Tím získáme dvoustupňové přenosové zřetězení, které je ilustrováno na obrázku 2.6.

Během stupně načtení instrukce je provedeno čtení z instrukční paměti nebo instrukční cache pro získání následující instrukce. Ve stupni dekódování jsou z instrukčního slova ex-



Obrázek 2.6: Schéma dvoustupňového přenosového zřetězení

trahovány zdrojové a cílové položky a přeposlány zásuvkám, které aktivují přenos dat do funkčních jednotek. Vlastní přenos dat se uskuteční ve stupni přesunu, ve kterém jsou data z výstupu funkčních jednotek přenesena do vstupních registrů jiných funkčních jednotek. Přístup do souborů registrů a předávání dat mezi funkčními jednotkami je taktéž prováděno ve stupni přesunu. Vedlejším efektem přesunů data může být spuštění operací. Možný prostor pro provádění těchto operací funkčními jednotkami je v taktech po stupni přesunu, což nás z pohledu přenosového zřetězení již nezajímá.

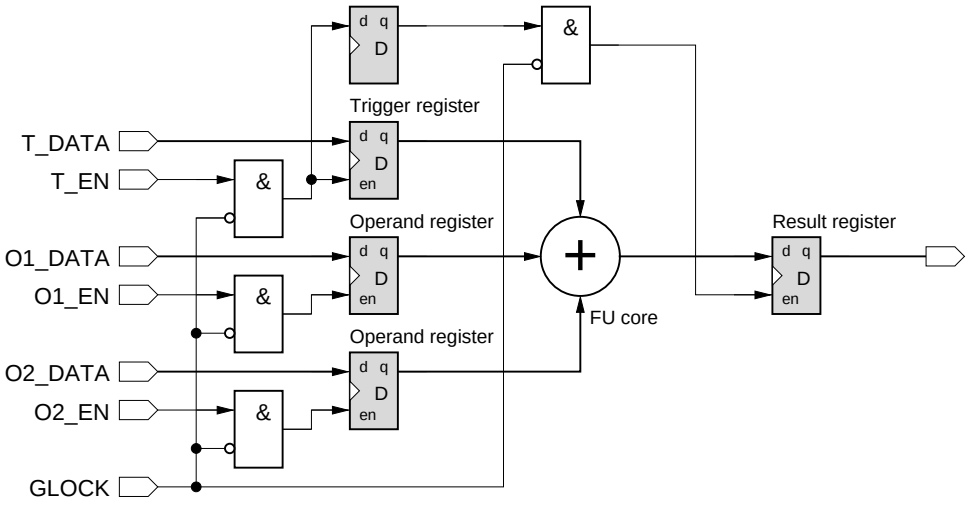
2.2.3 Funkční jednotky a soubory registrů

Funkční jednotky jsou komponenty, které provádějí výpočet a mohou komunikovat s externím prostředím. V TTA procesorech jsou obvykle přítomny jednotky pro načítání instrukcí, predikační jednotka[2] a jednotky nahrávání a ukládání dat. Jednotka načítání instrukcí čte instrukce z paměti a řídí tok programu. Je to jediná jednotka, které může ovlivňovat řídicí cestu procesoru. Jednotka načítání a ukládání dat poskytuje přístup k externí datové paměti.

Funkční jednotka obsahuje jeden nebo více vstupních a výstupních registrů. Vstupní registry mohou být dále rozlišeny na spouštěcí (*trigger*) a operandové (*operand*) registry. Výstupní registry se označují jako výsledkové (*result*) registry. Operandové registry uchovávají vstupní operandy funkční jednotky. Spouštěcí registry uchovávají vstupní operandy, ale také poskytují dvě další důležité funkce. Za prvé, Přenos do spouštěcího registru zahájí (*triggers*) novou operaci. Za druhé, pokud funkční jednotka podporuje více než jednu operaci, funkční jednotka ze zásuvky spolu s daty přijme operační kód (*opcode*), který vybírá operaci k provedení. Naprogramovaný přenos do operandového a spouštěcího registru se nazývá operandový přesun (*operand move*) a spouštěcí přesun (*trigger move*) v tomto pořadí. Podobně přenos, který čte výstup funkční jednotky, se nazývá výsledkový přesun (*result move*) pro danou funkční jednotku.

Podpora více vstupů a výstupů vyžaduje vícenásobné formáty instrukcí, které významně komplikují hardware instrukčního dekodéru jednoduchých RISC architektur. Architektury TTA jsou v tomto ohledu více flexibilní. Každý operand je v instrukci zakódován samostatně a do jednotky zapsán zvlášť, proto funkční jednotky nemají omezení na počet operandů. Tato vlastnost je obzvlášť důležitá pro aplikačně-specifické procesory, neboť umožňuje používat jednotky s jakýmkoliv počtem vstupních a výstupních operandů, aniž by se musela měnit obecná struktura procesoru.

[2]V originální anglické literatuře [1] se pro podmíněné vykonávání instrukcí používá termín *guarding*, což se dá přeložit jako strážení instrukcí. Problém s překladem nastává u výrazů pro označení podmínky vykonání *guard* (stráž?), *guard expression* (?), atd. Z tohoto důvodu se v této práci pro označení podmíněného vykonávání instrukcí přidržíme zavedeného pojmu *predikace*, používaného v české literatuře [4] z oblasti architektur procesorů.



Obrázek 2.7: Třívstupá jednotka sčítání používající SVTL řetězení

Operaci je vhodné zřetězit, pokud její provádění trvá více než jeden takt. Každá funkční jednotka může mít lokální zřetězenou linku s jedním nebo více stupni provádění operace, které jsou nezávislé na zřetězených linkách ostatních funkčních jednotek. V [1] je prezentováno několik disciplín zřetězení, které určují kdy je jednotlivým oddělujícím registrům linky dovoleno přijmout nová data. Mezi dvě nejpoužívanější metody řetězení patří *hybridní řetězení* a *řetězení virtuálního času*.

Funkční jednotky používající řetězení virtuálního času (VTL, *virtual-time latching*) běží synchronně s instrukčním proudem daným kompilátorem. S každým vydáním instrukce postoupí linka funkční jednotky o jeden krok. Pokud předpokládáme, že funkční jednotka je plně zřetězena, tj. počet stupňů linky je roven její latenci, pak stupeň přijme nová data s každým vydáním instrukce. Odpovědností kompilátoru je zajistit, aby nedošlo k přepsání dále ještě potřebných dat. Linka je pozastavena pouze při čekacích taktech (*stalls*), které mohou být způsobeny např. výpadkem instrukční cache. Existují dvě rozdílné verze řetězení virtuálního času, které se liší v tom, jak se zachází s přenosy do operandových registrů.

U řetězení pravého virtuálního času (TVTL, *true virtual-time latching*) každý přesun do vstupních registrů odstartuje novou operaci. To činí hardware jednodušším, ale omezuje to svobodu plánování kompilátoru. U řetězení téměř virtuálního času (SVTL, *semi virtual-time latching*) pouze spouštěcí přesun aktivuje operaci funkční jednotky. Pro každý operandový registr musí být přidán stínový registr, který dočasně ukládá přednačtenou hodnotu operandu do doby zahájení operace. Nicméně stínové registry mohou být vynechány pokud má funkční jednotka kromě operandových a spouštěcího registru ještě další registrované stupně. Obrázek 2.7 zobrazuje třívstupou jednotku sčítání používající SVTL řetězení. Vstupní registry načítají data z odpovídající vstupní zásuvky, pokud je povolovací vstup aktivní. Jádro funkční jednotky, v tomto případě třívstupá sčítačka, provádí operaci na registrovaných hodnotách dvou operandových a jednoho spouštěcího registru. Výsledkový registr je aktualizován, pokud řídicí signál `T_EN` byl aktivní v předchozím taktu. Latence sčítací jednotky je tudíž dva takty, což znamená, že hodnota z výsledkového registru může být vyčtena po dvou taktech od spuštění operace. Výsledná hodnota zůstává k dispozici do té doby, než je přepsána výsledkem další operace. Jelikož je funkční jednotka zřetězena, nová operace může být spouštěna každý takt. Linka funkční jednotky je pozastavena, pokud je aktivní signál globálního blokování `GLOCK`.

Hybridní řetězená linka načítá data tehdy, pokud registry linky neobsahují platná data z předchozí zahájené operace, která už dále nepokračuje v lince. To je zajištěno připojením řadiče zřetězeného stupně ke každému registru zřetězené linky funkční jednotky. Poslední stupeň linky pokračuje pouze tehdy, když je k funkční jednotce přistoupeno pomocí výsledkového přesunu. Z tohoto důvodu musí být linka funkční jednotky vyprázdněna, pokud byla spuštěna spekulativní operace, jinak nemůže být spuštěna další operace a dojde k zablokování jednotky.

V [5] byly porovnány tyto dvě alternativy řetězení z pohledu celkového počtu cyklů. Přestože hybridní řetězení poskytuje kompilátoru větší svobodu plánování v porovnání s řetězením virtuálního času, požadované vyprazdňovací přesuny degradují výkonnost hybridního řetězení, pokud je aplikováno spekulativní vykonávání kódu. Při použití hybridního řetězení na dané sadě srovnávacích aplikací se nenašla žádná významná plánovací výhoda. Proto je obecně preferováno řetězení virtuálního času pro jeho jednoduchou řídicí logiku.

Procesory TTA vyžadují registry obecného určení (GPR, *general-purpose registers*) pro ukládání mezivýsledků. Registry obecného určení jsou uspořádány jako jeden nebo více souborů registrů, které jsou připojeny k propojovací síti přes vstupní a výstupní zásuvky, stejně jako normální funkční jednotky. V architekturách TTA může být počet portů souboru registrů značně zredukován v porovnání s VLIW architekturami. Kromě toho může být soubor registrů s větším počtem portů efektivně rozdělen do více souborů registrů s menším počtem portů bez citelnější ztráty výkonu. [6, 7]

2.3 Softwarové aspekty

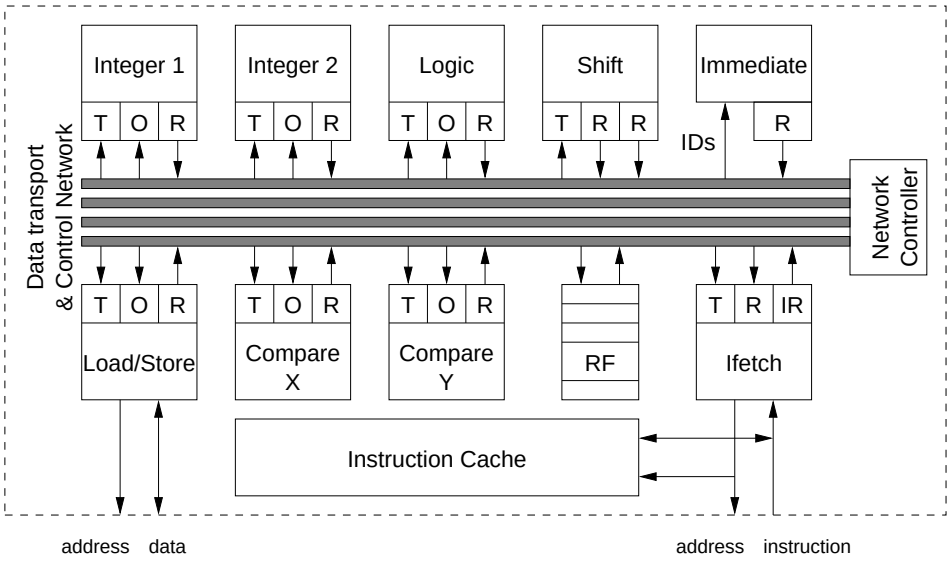
Program pro tradiční *operací spouštěné architektury* (OTA, *Operation Triggered Architectures*), jako jsou architektury RISC a VLIW, se skládá z uspořádané množiny operací, které jsou vykonávány procesorem. Jak již bylo zmíněno, tak přenosem spouštěné architektury jsou programovány uvedením přenosů dat mezi funkčními jednotkami a soubory registrů přes propojovací síť. Z tohoto důvodu je k dispozici pouze jediná instrukce – instrukce přesunu dat (*move*). Proto se také někdy TTA architektury nazývají jako MOVE architektury a jejich realizace jako MOVE procesory.

Pro ilustraci principu programování TTA architektur si ukážeme, jak se RISCová instrukce převede na posloupnost přesunů dat TTA procesoru:

```
                          r1 -> add.o;
add r3, r2, r1   ⇒        r2 -> add.t;
                          add.r -> r3;
```

Cíle přesunu `add.o` a `add.t` označují operandový a spouštěcí registr sčítačky a zdroj `add.r` označuje výsledkový registr (výstup) sčítačky. Pro zajištění plné programovatelnosti TTA architektur potřebujeme podporu pro řízení toku programu a podmíněné vykonávání kódu. Operace řízení toku programu, jako jsou skoky a volání, mohou být implementovány zpřístupněním programového čítače v jednotce načítání instrukcí jako možný zdroj a cíl přesunů v programu. Podmíněné vykonávání kódu může být uskutečněno predikací operací přesunů. Přenosy dat se pak uskuteční za základě splnění predikátu.

Způsob programování TTA architektur se může na první pohled zdát těžkopádný. Namísto uvedení jedné dyadické operace musí být uvedeny tři přesuny. Nicméně tento přístup poskytuje příležitosti pro mnoho optimalizací v době kompilace, které nejsou možné u tradičních architektur. Protože všechny přenosy mohou být nyní řízeny programátorem nebo



Obrázek 2.8: Funkční pohled na procesor MOVE32INT

kompilátorem, mohou být zbytečné přenosy, které se často vyskytují ve stávajících architekturách, ušetřeny. To je klíčové z pohledu řešení problému škálovatelnosti VLIW achitektur. [1]

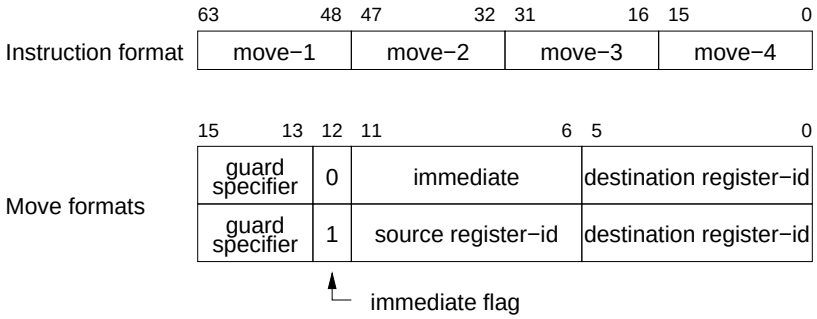
Programování a optimalizování kódu pro TTA procesory je významně složitější v porovnání s tradičními OTA procesory. Psaní programu ručně by bylo strašně zdlouhavé a náchylné na chyby. Proto se programy popisují v jazycích vyšší úrovně např. C nebo Java. Kompilátor transformuje vysokoúrovňový kód na přesuny dat mezi hardwarovými prostředky a plánovač instrukcí optimalizuje kód s cílem minimalizovat dobu běhu a velikost kódu. Podrobnosti o generování kódu a optimalizacích jsou diskutovány v [3, 5].

2.4 Realizace TTA procesoru MOVE32INT

Aby mohly být vyhodnoceny konkrétní kompromisy při návrhu a implementaci, byla na Delft University of Technology navržena instance TTA architektury nazývaná MOVE32INT [2]. Procesor se skládá především z přenosové sítě řízené řadičem sítě a několika funkčních jednotek. Přenosová síť obsahuje čtyři sběrnice. Každá sběrnice je složena z datové sběrnice schopné přenášet datové hodnoty o šířce 32 bitů, identifikační sběrnice přenášející jeden specifikátor přesunu délky 16 bitů (viz obrázek 2.9) a řídicí sběrnici obsahující několik řídicích signálů. Paralelně je možné provádět čtyři datové přenosy.

Obrázek 2.8 ukazuje pohled na procesor MOVE32INT z funkčního pohledu, kde jsou znázorněny operandové (R), spouštěcí (T) a výsledkové registry (R) funkčních jednotek. Jak je patrno, tak procesor používá harvardskou architekturu, tj. že má oddělené paměťové sběrnice pro instrukce a data. Procesor MOVE32INT obsahuje 10 funkčních jednotek.

Celočíselné jednotky (*integer*) provádějí sčítání a odečítání. Logická jednotka provádí logické operace AND, OR a XOR nad dvěma operandy. Jednotka posuvů (*shift*) provádí jednobitové a dvoubitové posuvy na datech načtených do spouštěcího registru a ukládá jednotlivé výsledky do samostatných výsledkových registrů. Dále zde je jedna jednotka přímých operandů pro každou sběrnici. Čtení z této jednotky má za účinek vložení 6-bitového zdrojového identifikátoru operace přesunu jako neznaménkově rozšířenou hodnotu na jednu ze



Obrázek 2.9: Instrukční formát a formát přesunu procesoru MOVE32INT

sběrnic. Porovnávací jednotky (*compare*) produkují příznak, který může být vyčten z výsledkových registrů. Jejich výstup není připojen na datovou sběrnici, ale na řídicí sběrnici a je použit k predikaci datových přenosů. Jednotka přístupu do paměti (*load/store*) obsahuje spouštěcí a výsledkový registr pro načítání a spouštěcí a operandový registr pro ukládání. Jednotka je připojena na externí datovou paměť. Jednotka načítání instrukcí (*ifetch*) je v podstatě jednotka pro načítání dat s tím, že adresa (programový čítač) je automaticky inkrementován. Zápis do spouštěcího registru způsobí přepsání programového čítače a tím pádem dojde i ke skoku v programu. Čtení z výsledkového registru vrátí adresu stávající instrukce plus 2. Jednotka načítání instrukcí je připojena k malé interní instrukční cache. Funkční jednotky jsou implementovány pomocí hybridního řetězení. Transportní řetězení procesoru MOVE32INT používá třístupňovou transportní linku, která byla zmíněna v kapitole 2.2.2.

Každá instrukce procesoru MOVE32INT obsahuje 64 bitů a specifikuje čtyři přenosy. Formát instrukce a formát specifikace přesunu je zobrazen na obrázku 2.9. Obsahuje tříbitový predikační identifikátor, jednobitový příznak přímého operandu a 6-bitové zdrojové a cílové položky. Příznak přímého operandu určuje interpretaci zdrojové položky. Pokud je nastaven, zdrojová položka obsahuje krátký 6-bitový přímý operand, jinak obsahuje identifikátor zdrojového registru.

Procesor MOVE32INT umožňuje velmi obecnou podporu podmíněného vykonávání kódu. Každý přesun může být podmíněně vykonán. Podmínka je určena tříbitovým predikačním identifikátorem u každého přesunu. Během každého taktu jsou vytvořeny čtyři predikáty, pro každou sběrnici jeden predikát. Predikát určuje, zda bude přesun uskutečněn nebo bude potlačen (*squashed*). Predikační identifikátor říká, jakým způsobem vyhodnotit predikát. Predikát může být tvořen z více booleovských výrazů daných výsledkem porovnávacích jednotek.

Procesor MOVE32INT byl realizován a vyroben pomocí 2 μm (minimální délka hradla 1,6 μm, 2 μm metalické vrstvy) CMOS technologie Sea of Gates (SoG). Obraz SoG obsahuje 88 řádků, každý řádek má 1088 tranzistorových párů, což celkem činí 191 tisíc tranzistorů. Celková velikost matrice je 1×1 cm^2. MOVE32INT dosahuje poměrně vysoké pracovní frekvence 80 MHz, nazvzdory nenáročnosti použité technologie.

Kapitola 3

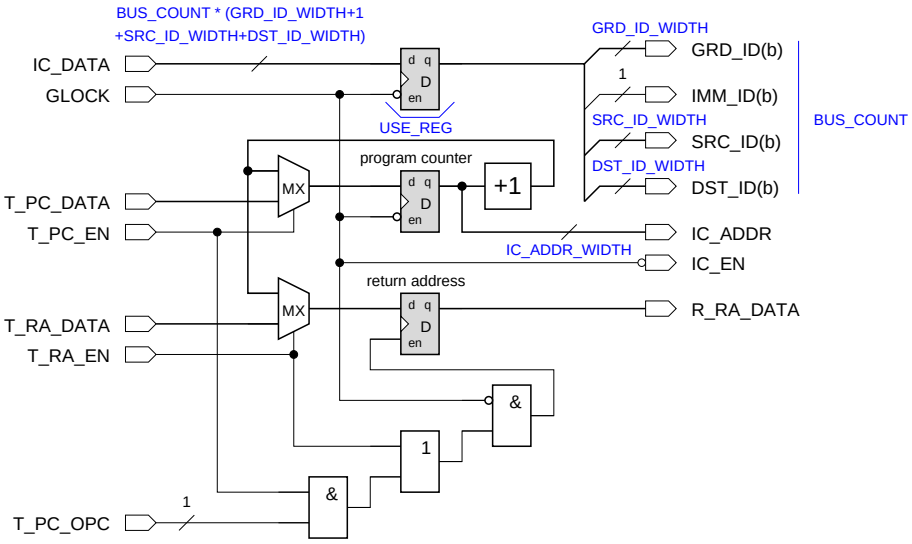
Návrh a implementace generických komponent TTA architektury

Tato kapitola se zabývá návrhem generických komponent TTA architektury. Vhodným propojením těchto komponent je v kapitole 4 realizován ukázkový generický TTA procesor GENTTA. Implementace je provedena v jazyce VHDL s ohledem na syntézu do programovatelného hradlového pole FPGA. Realizace bude ověřena konkrétně pro hradlové pole Virtex-II Pro [12] od firmy Xilinx. Pro popis generických jednotek byl zvolen behaviorální styl popisu, který je kompaktnější než strukturální popis. Při implementaci bylo dbáno na to, aby nebyl přímo instancován konkrétní logický člen, který je dostupný pouze na jedné platformě FPGA. Z tohoto důvodu by mělo být snadné portovat navržené jednotky i pod hradlové pole jiného výrobce.

V maximální možné míře jsou možné volitelné záležitosti parametrizovány pomocí generických parametrů. Jedná se především o datovou šířku procesoru, šířky adresových sběrnic, ale i různé příznaky, které zapínají nebo vypínají určitou funkčnost designu, jako je povolení předávací logiky, přidání výstupních registrů, atd.

Jazyk VHDL obecně podporuje datový typ pole. Např. pro standardní datový typ std_logic představující jeden bit je k dispozici datový typ std_logic_vector, kterému může být při deklaraci uvedena požadovaná datová šířka. Problém nastává, pokud chceme mít v rozhraních jednotek generický počet portů, např. generickým parametrem určit počet čtecích portů souboru registrů. VHDL umožňuje nadefinovat pole obecně jakéhokoliv typu, ale při deklaraci signálu nebo proměnné umožňuje parametrizovat pouze vnější rozměr, tj. počet prvků pole, ale už neumožňuje generickým parametrem ovlivnit, jakou šířku mají mít položky. Řešení se nabízí v použití datového typu matice, kde můžeme generickým parametrem určit oba rozměry. Bohužel je pak složitějším získávání jednotlivých řádků matice, protože se matice musí adresovat jako dvourozměrné pole. Navíc ne všechny překladové nástroje podporují tyto ne zcela běžně používané datové typy.

Částečným řešením, které je použito i v této práci, je pole stejně dlouhých vektorů rozložit do jednoho dlouhého vektoru. Toto řešení podporují všechny překladové nástroje, ale problémem zůstává, že tímto převodem ztratíme informaci o velikosti obou rozměrů pole a musíme mít tuto informaci uloženou někde bokem, např. pomocí konstant nebo generických parametrů. Při tomto přístupu se stává složitější i přistupování k jednotlivým vektorům. Nejčastější chybou je, že dojde k záměně počtu a šířky prvků pole a tyto chyby se hůře hledají, neboť nutně nemusí dojít k chybové hlášce při překladu. Nicméně při udržování konvencí se dá tento přístup s úspěchem použít.



Obrázek 3.1: Blokové schéma jednotky načítání instrukcí

V této práci se přidržujeme konvence, že údaj o velikosti pole stojí vlevo a údaj o šířce prvků vpravo. Například u deklarace portu v rozhraní entity

```
SRC_DATA : in  std_logic_vector(SRC_COUNT*DATA_WIDTH-1 downto 0);
```

`SRC_COUNT` udává velikost a `DATA_WIDTH` šířku prvků. Nicméně v tomto příkladě je to patrné i podle názvů.

3.1 Řídicí jednotka

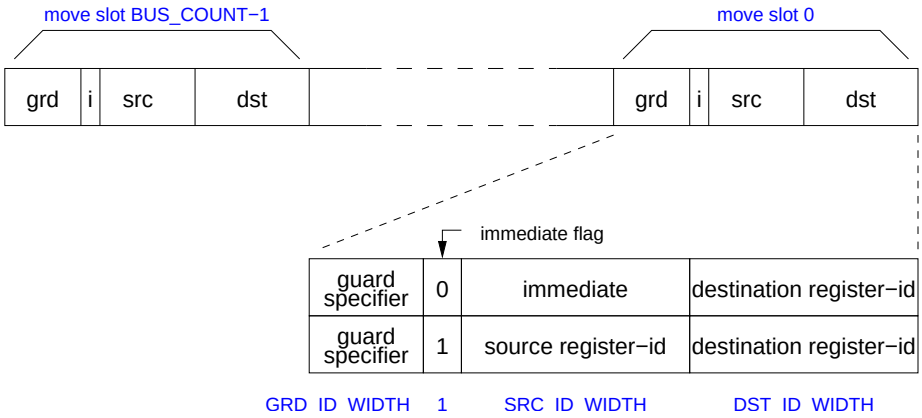
Řídicí jednotka je společné označení pro sadu jednotek, bez kterých se neobejde žádný TTA procesor. Mezi tyto jednotky patří jednotka načítání instrukcí, jednotka dekódování instrukcí, jednotka přímých operandů a predikační jednotka.

3.1.1 Jednotka načítání instrukcí

Jednotka načítání instrukcí (IFU, *Instruction Fetch Unit*) načítá instrukce z paměti. Jednotka obsahuje programový čítač (PC, *program counter*), který je automaticky inkrementován v každém taktu, pokud není přepsán přímým přiřazením hodnoty do spouštěcího registru `T_PC_DATA` při skoku nebo volání v programu. Aktuální hodnota programového čítače je použita jako adresa do instrukční paměti. Pokud nastane skok, je potřeba změnit pouze obsah programového čítače, zatímco při volání musí být návratová adresa (hodnota programového čítače plus jedna) uložena do registru návratové adresy (RA, *return address*). Rozlišení operace se děje pomocí jednobitového operačního kódu. Jednotka načítání instrukcí je zobrazena na obrázku 3.1.

Instrukční registr volitelně ukládá instrukční slovo `IC_DATA` načtené z instrukční paměti. Tento registr se použije, pokud je nastaven generický parametr `USE_REG`. Jeho potlačení je užitečné v případech, kdy čtení z instrukční paměti nebo cache je synchronní.

Podporovaný instrukční formát můžete vidět na obrázku 3.2. Instrukční formát je generický, tzn. může být ovlivněn generickými parametry. Generický parametr označující



Obrázek 3.2: Obecný instrukční formát procesoru GENTTA

počet sběrnic `BUS_COUNT` udává, kolik současných přenosů je podporováno. Přenos se skládá z predikačního identifikátoru o generické šířce `GRD_ID_WIDTH`, příznaku přímého operandu, který udává, zda zdrojová položka přesunu obsahuje krátký přímý operand nebo identifikátor zdrojového registru. Přímý operand nebo identifikátor zdrojového registru má šířku `SRC_ID_WIDTH`. Poslední položkou je identifikátor cílového registru o šířce `DST_ID_WIDTH`.

3.1.2 Jednotka dekódování instrukcí

Jednotka dekódování instrukcí (IDU, *Instruction Decode Unit*) z identifikátorů obsažených v instrukčním slově dékóduje řídicí signály pro propojovací síť a funkční jednotky. Schéma jednotky je naznačeno na obrázku 3.3. Funkce jednotky je logicky rozdělena na tři části: dekódování řídicích signálů pro podporu krátkých přímých operandů, dekódování signálů pro zdrojové zásuvky a dekódování signálů pro cílové zásuvky.

Dekódování přímých operandů – vstupem jsou jednobitové příznaky přímého operandu `IMM_ID`, které určují, zda zdrojová položka v instrukci obsahuje přímý operand nebo adresu registru. Pro každou sběrnici je jeden příznak. Dekódování je velice přímočaré, a pro každý identifikátor je proveden logický součin s negací odpovídajícího potlačovacího signálu `SQUASH`, jejichž počet je dán opět počtem sběrnic. Tím získáme řídicí signály `IMM_EN` pro propojovací síť, které můžeme na výstupu volitelně registrovat (určeno generickým parametrem `USE_REG`).

Dekódování zdrojových zásuvek – na obrázku 3.3 uprostřed vidíme dekódovací logiku pro jednu zdrojovou zásuvku. Celkový počet zdrojových zásuvek je dán generickým parametrem `SRC_COUNT`. Vstupem jsou zdrojové identifikátory, pro každou sběrnici jeden, které jsou porovnány s napevno zadrátovanou konstantou[1], která představuje adresu zdrojové zásuvky. Výsledek porovnání je vymaskován potlačovacím signálem `SQUASH` a identifikátorem přímého operandu, jelikož při jeho nastavení zdrojová položka neobsahuje platnou adresu zásuvky, ale krátký přímý operand. Takto vymaskované bity tvoří přímo výběrový signál `SRC_SEL` pro propojovací síť. Použité kódování výběrového signálu stylem one-hot je důsledkem toho, že na více sběrnicích může být současně požadavek na čtení z jedné zdrojové zásuvky. To je povoleno a nevadí nám

[1]Díky nemožnosti definovat generické parametry o generické šířce v jazyce VHDL, se ve skutečnosti musí pro adresu zásuvky použít normální port, na který se připojí konstantní hodnota při instancování jednotky.

to, pokud je na všech sběrnicích shodná nejen adresa zdrojové zásuvky, ale i operační kód. V opačném případě se dostáváme do konfliktu, jelikož pro přenesení operačního kódu `SRC_OPC` máme pouze jednu sběrnici a nevíme, z které sběrnice máme operační kód vybrat. K řešení tohoto konfliktu se nabízí prioritní kóder[2], který nám na základě výběrových signálů vrátí binárně zakódovaný selektor pro multiplexor, který vybere patřičný operačními kód. Operační kód je vlastně celý zdrojový identifikátor, jelikož nemáme pevně stanovenou hranici, kde začíná a kde končí adresa zásuvky a operační kód. Starost o nekonfliktní zakódování adres zásuvek a operačních kódu je ponecháno na návrháři procesoru. Funkční jednotka musí z operačního kódu použít jen odpovídající bity. Povolovací signál pro zápis do registru `SRC_EN`, který je přiveden přímo k funkční jednotce, se získá logickým součtem výběrových signálů. Všechny dekódované signály mohou být na výstupu volitelně registrovány.

Dekódování cílových zásuvek – na obrázku 3.3 dole vidíme dekódovací logiku pro jednu cílovou zásuvku. Na první pohled je patrné, že je téměř shodná s logikou dekódování zdrojových zásuvek až na několik málo změn. První změnou je to, že se nevymaskovává podle příznaků přímých operandů jako v předchozím případě, jelikož cílové položky v instrukci nemohou specifikovat přímý operand. Druhou a tou podstatnější změnou je to, že do jedné cílové zásuvky nemůže posílat data z více sběrnic. Pokud nastane konflikt, může pro výběrový signál `DST_SEL` použít stejný prioritní kodér jako pro přepínání operačních kódů. Pro M sběrnic stačí k binárnímu zakódování výběrového signálu $\lceil log_2 M \rceil$ bitů.

3.1.3 Jednotka přímých operandů

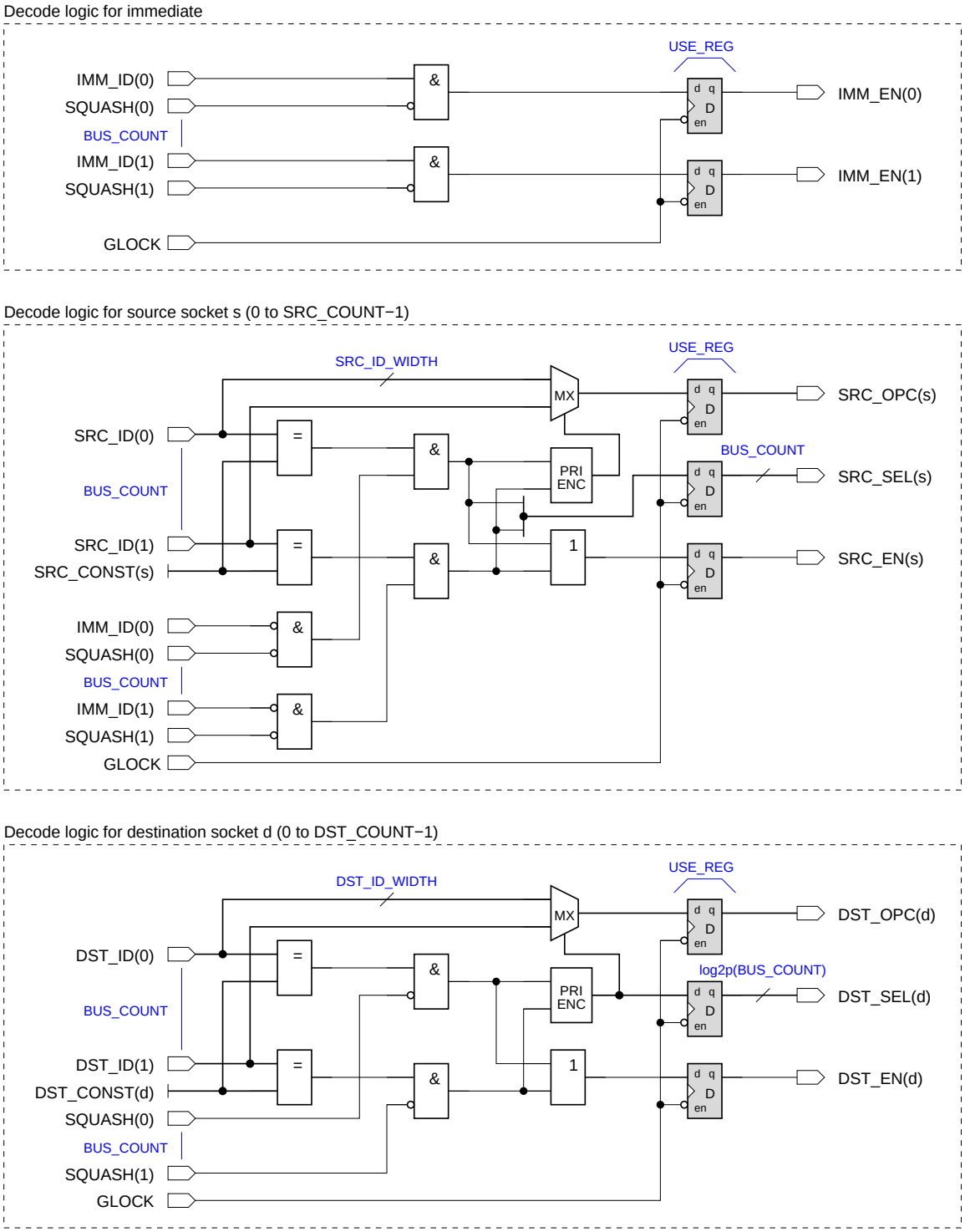
Zdrojové identifikátory mohou být použity k uchování krátkých přímých operandů, pokud je nastaven odpovídající příznak přímého operandu v instrukčním slově. Jednotka přímých operandů (IMMU, *Immediate Unit*) zajišťuje znaménkové/beznaménkové rozšíření krátkých přímých operandů na datovou šířku procesoru a jejich uložení do registrů. Jednotka přímých operandů je zobrazena na obrázku 3.4 a její rozhraní je popsáno v tabulce A.2.

Počet přímých operandů je dán počtem sběrnic (generický parametr `BUS_COUNT`). Krátký přímý operand `SRC_ID` má šířku určenou generickým parametrem `SRC_ID_WIDTH`. Pokud je nastaven generický parametr `SGN_EXT`, dojde ke znaménkovému rozšíření krátkého přímého operandu na datovou šířku procesoru, určenou generickým parametrem `DATA_WIDTH`, zkopírováním MSB bitu krátkého operandu, v opačném případě se přímý operand `IMM_DATA` doplní logickými nulami (beznaménkové rozšíření). Pokud je nastaven generický parametr `USE_REG`, jsou všechny přímé operandy na výstupu registrovány. Zápis do výstupních registrů je povolen, pokud není aktivní signál globálního blokování `GLOCK`.

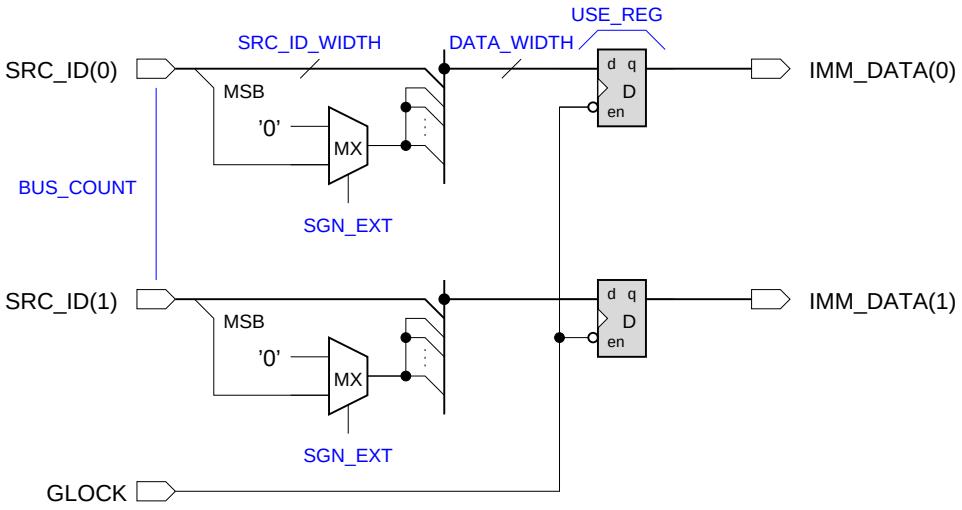
3.1.4 Predikační jednotka

Predikační jednotka (GU, *Guard Unit*) vyhodnocuje predikační výraz, který je dán predikačním identifikátorem `GRD_ID`. Tato jednotka umožňuje podmíněné vykonávání kódu. Pokud je daný predikační výraz vyhodnocen jako nepravda, odpovídající přenos na sběrnici je zrušen. Zrušené přenosy jsou indikovány jednotce IDU pomocí tzv. potlačovacích

[2]V této práci se přidržujeme toho, že největší prioritu má vždy sběrnice s nejnižším číslem, tj. sběrnice s číslem 0, pokud není explicitně uvedeno jinak.



Obrázek 3.3: Blokové schéma jednotky dekódování instrukcí



Obrázek 3.4: Blokové schéma jednotky přímých operandů

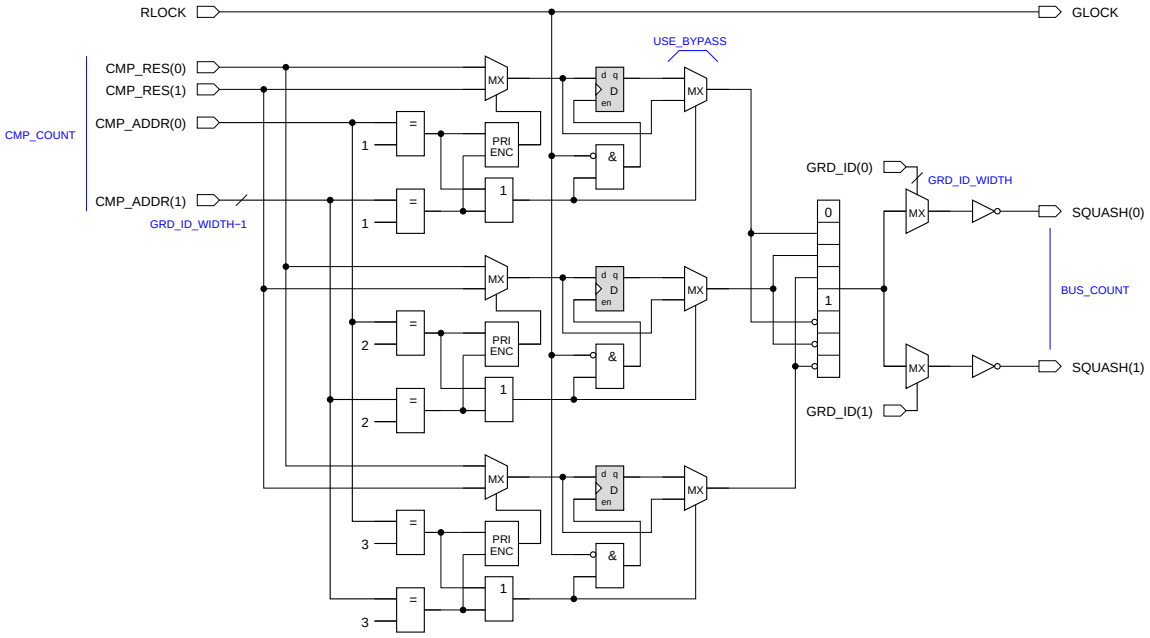
Bin. hodnota	**Výraz**	**Interpretace**
000	false	přesun nikdy nevykonej
001	b_1	vykonej přesun, pokud booleovských registr 1 obsahuje '1'
010	b_2	vykonej přesun, pokud booleovských registr 2 obsahuje '1'
011	b_3	vykonej přesun, pokud booleovských registr 3 obsahuje '1'
100	true	přesun vždy vykonej
101	$\neg b_1$	vykonej přesun, pokud booleovských registr 1 obsahuje '0'
110	$\neg b_2$	vykonej přesun, pokud booleovských registr 2 obsahuje '0'
111	$\neg b_3$	vykonej přesun, pokud booleovských registr 3 obsahuje '0'

Tabulka 3.1: Kódování predikačních výrazů pro tříbitový predikační identifikátor

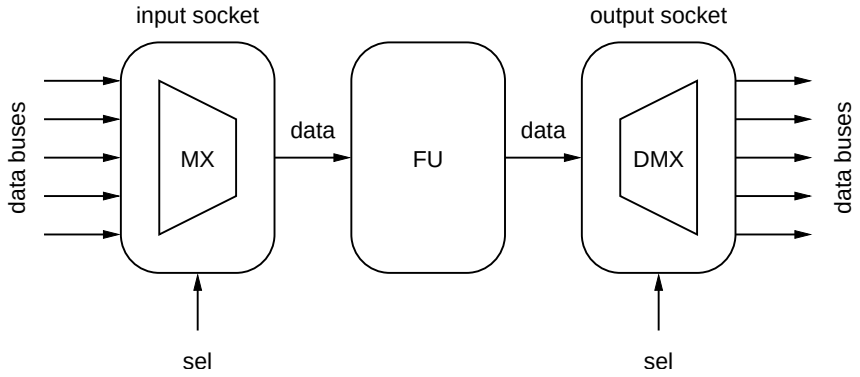
signálů (*squash signals*). Dále jednotka obsahuje booleovské registry, které uchovávají výsledky porovnání z porovnávacích funkčních jednotek. Predikační jednotka může taky vyvolat pozastavení procesoru, což je propagováno jako globální blokovací signál `GLOCK` ke všem zřetězeným jednotkám procesoru. Umožňuje to pozastavit zřetězenou linku na určitý počet taktů. V našem případě jednotka pouze propaguje signál žádosti o blokování `RLOCK`, který je vně predikační jednotky. Obrázek 3.5 zobrazuje strukturu predikační jednotky.

Predikační jednotka obsahuje pole bitových registrů, jejich počet je pro šířku predikačního identifikátoru GW dán jako 2^{GW-1} mínus jeden registr. Chybějící registr s číslem nula má vždy konstatní hodnotu logické nuly a slouží k realizaci nepodmíněného vykonávání kódu. Predikační výraz umožňuje použít jako predikát hodnotu některého z registrů, případně jeho negaci. Šířku predikačního identifikátoru ovlivňuje generický parametr `GRD_ID_WIDTH`. Příklad predikačních výrazů pro tříbitový predikační identifikátor je uveden v tabulce 3.1.

Predikační jednotka umožňuje připojení `CMP_COUNT` porovnávacích jednotek. Výsledek porovnání `CMP_RES` se zapíše na adresu bitu danou adresou `CMP_ADDR`, zápis na adresu nula se ignoruje. Pokud je připojeno více porovnávacích jednotek a ty zapisují současně na stejnou adresu, musí být tento konflikt řešen pomocí priority porovnávací jednotky s nižším číslem. Z jednotlivých registrů je sestaven predikační výraz, z jehož predikátů se vybírá pomocí predikačního identifikátoru a výsledek se zneguje a získá se potlačovací



Obrázek 3.5: Blokové schéma predikační jednotky

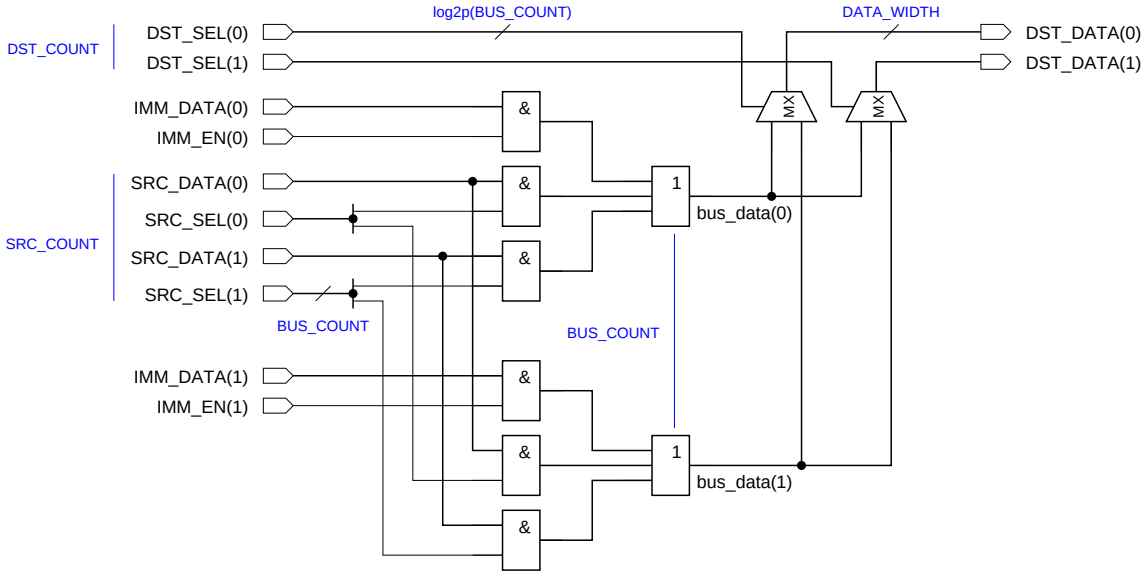


Obrázek 3.6: Konceptuální pohled na vstupní a výstupní zásuvky

signál `SQUASH`. Pro každou sběrnici je potřeba jeden potlačovací signál. Pokud má být do pole bitových registrů umožněn čtení a zápis z/do jednoho registru v stejném taktu, je potřeba v případě shody čtecí a zápisové adresy provést předání (*bypass*) hodnoty pomocí multiplexoru. Tato možnost se dá zapnout pomocí generického parametru `USE_BYPASS`.

3.2 Propojovací síť

Propojovací síť se skládá ze sběrnic a zásuvek, jak již bylo naznačeno na obrázku 2.4. Zásuvky poskytují programovatelné propojení ke sběrnicím. Vstupní zásuvky jsou v podstatě multiplexorem, který vybírá hodnotu z jedné sběrnice a zapisuje do vstupního registru funkční jednotky. Výstupní zásuvky jsou naopak demultiplexory, které přenášejí obsah výsledkového registru funkční jednotky na jednu ze sběrnic. Konkrétní implementace sběrnic a zásuvek může být provedena pomocí AND-OR struktury, třístavových sběrnic nebo multiplexovaných sběrnic.



Obrázek 3.7: Propojovací síť realizovaná pomocí AND-OR struktury

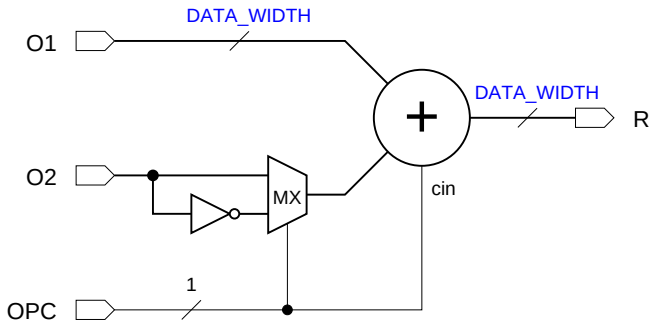
Obrázek 3.6 zobrazuje koncepční pohled na činnost zásuvek v propojovací síti pro 5-sběrnicový TTA procesor. Vstupní zásuvka vybírá propojení odpovídající sběrnice na vstupní registr funkční jednotky, což je řízeno pomocí řídicího signálu z řídicí jednotky. Podobně i výstupní zásuvka vybírá sběrnici, na kterou bude připojen datový výstup funkční jednotky.

Obrázek 3.7 ukazuje realizaci propojovací sítě pomocí AND-OR struktury pro dvousběrnicovou propojovací síť se dvěma zdrojovými a cílovými zásuvkami. Na přímé operandy může být nahlíženo jako na jednu ze zdrojových zásuvek. Výběrové signály `SRC_SEL` z dekódovací jednotky jsou použity k získání logických součinů s odpovídajícími datovými signály `SRC_DATA`. Aktivní může být pouze jedno AND hradlo na sběrnici, ostatní jsou v logické nule. Logickým součtem pak dosáhneme propojení aktivního AND hradla na sběrnici. Data výstupních zásuvek jsou vybírána pomocí multiplexorů z datových sběrnic, výběr je řízen binárně kódovaným výběrovým signálem `DST_SEL`.

3.3 Knihovna funkčních jednotek

Tato kapitola si klade za cíl navrhnout a popsat sadu funkčních jednotek využitelných pro realizaci obecného procesoru. Funkční jednotka je uživatelsky definovaná logika se standardním rozhraním. Standardní rozhraní je tvořeno spouštěcími, operandovými a výsledkovými registry. Konkrétní signály tohoto rozhraní byly prezentovány na již uvedeném obrázku 2.7. Funkční jednotky mohou vykonávat různé operace a mohou komunikovat s externími zařízeními, např. s pamětí RAM. Funkční jednotky mohou např. provádět násobení, bitové posuny nebo načítat data z paměti. TTA procesor je typicky složen z více různých jednotek, aby mohl vykonávat požadované operace.

Všechny funkční jednotky jsou implementovány pomocí disciplíny zřetězení SVTL (viz kapitola 2.2.3. Jelikož většina prezentovaných jednotek je tvořena kombinační logikou, nemá význam u každé jednotky opakovaně uvádět podrobnosti o připojení komunikačních registrů. To již bylo ukázáno na obrázku obrázku 2.7. V dalším textu se zaměříme jen na



Obrázek 3.8: Blokové schéma kombinační části aritmetické jednotky

Opcode	Název	Popis
0	ADD	Aritmetický součet ($R = O1 + O2$).
1	SUB	Aritmetický rozdíl ($R = O1 - O2$).

Tabulka 3.2: Podporované operace aritmetické jednotky

kombinační jádro jednotek.

3.3.1 Aritmetická jednotka

Na obrázku 3.8 je zobrazeno kombinační jádro aritmetické jednotky (AU, *Arithmetic Unit*), která podporuje operace aritmetické součtu a rozdílu v pevné řádové čárce (viz tabulka 3.2).

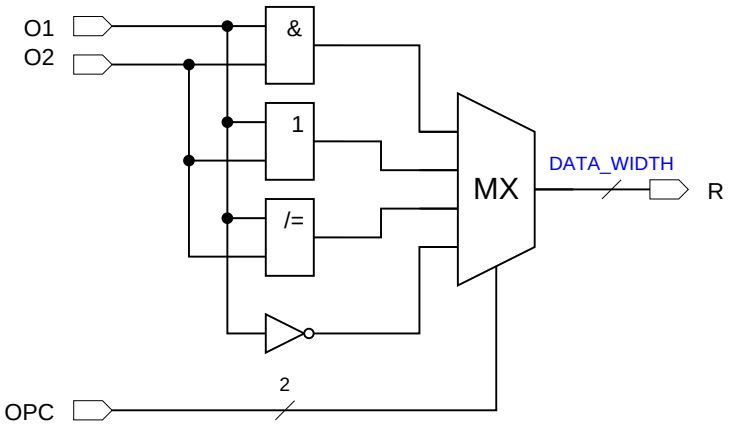
Operace sčítání a odečítání by mohly být implementovány v samostatných jednotkách pomocí sčítačky a odečítačky. To by umožňovalo v případě potřeby provádět tyto operace paralelně. Nevýhodou je, že takovéto řešení vyžaduje dvojnásobné množství hardwarových zdrojů i v případě, že nikdy obě jednotky nevyužijeme paralelně. Existuje jednoduchý postup, jak převést operaci odečítání na operaci sčítání a sloučit je do jedné jednotky, bez výrazného dopadu na hardwarové zdroje. Operace odečítání lze jednoduše převést na operaci sčítání tak, že se z operandu vytvoří dvojkovový doplněk. Dvojkový doplněk je pak realizován negací operandu s přičtením jedničky. Multiplexor vybírá mezi normální a negovanou hodnotou operandového registru. K řízení multiplexoru je použit operační znak, který je zároveň i vstupním přenosem do sčítačky.

3.3.2 Logická jednotka

Na obrázku 3.9 je zobrazeno schéma kombinační logiky logické jednotky (LU, *Logic Unit*), která podporuje běžné logické operace jako AND, OR, XOR a NOT. Úplný seznam operací a popis včetně rovnice je uveden v tabulce 3.3.

3.3.3 Jednotka bitových posunů a rotací

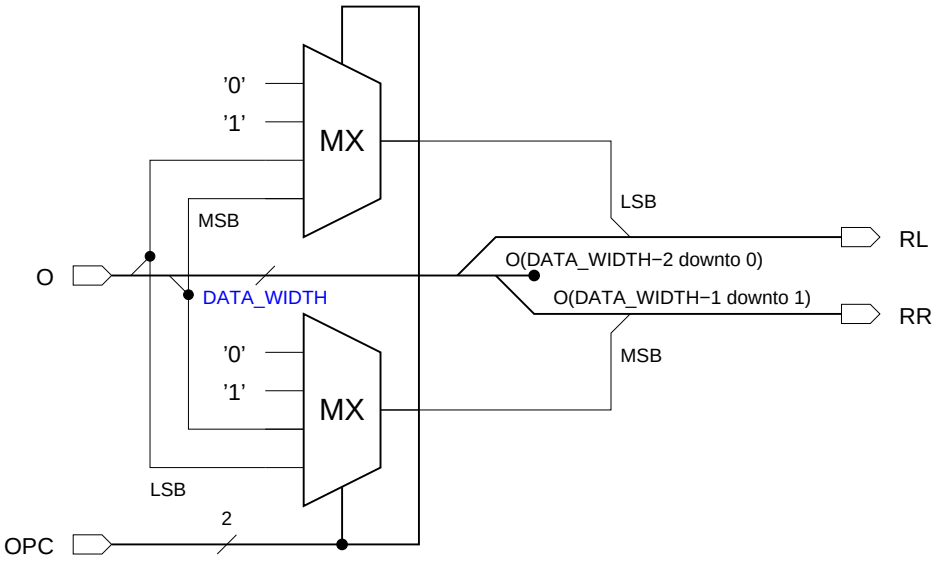
Na obrázku 3.10 je zobrazeno schéma kombinační logiky jednotky bitových posunů a rotací (SRU, *Shift and Rotate Unit*). Jednotka podporuje bitové posuny a rotace o jeden bit. U bitových posunů je podporováno nasouvání logické 0, logické 1 a významových bitů (LSB, MSB). Pro každý směr posunu/rotace (doleva a doprava) je vytvořen samostatný výsledek. Seznam všech operací a jejich popis je uveden v tabulce 3.4.



Obrázek 3.9: Blokové schéma kombinační části logické jednotky

Opcode	**Název**	**Popis**
00	AND	Logický součin ($R = O1 \wedge O2$).
01	OR	Logický součet ($R = O1 \vee O2$).
10	XOR	Exkluzivní logický součet ($R = O1 \oplus O2$).
11	NOT	Negace ($R = \overline{O1}$).

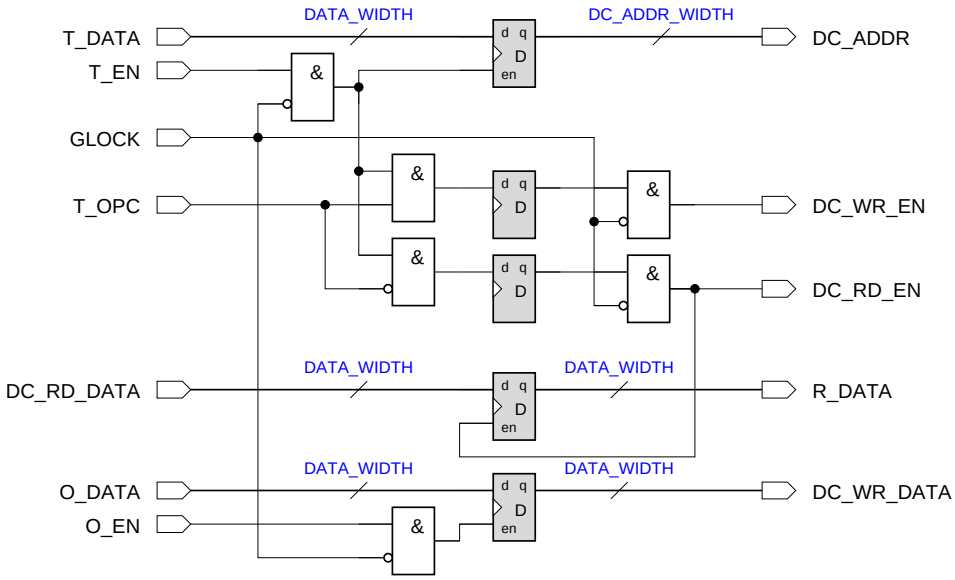
Tabulka 3.3: Podporované operace logické jednotky



Obrázek 3.10: Blokové schéma kombinační části jednotky bitových posunů a rotací

	Opcode	Název	Popis
RL	00	SL0	Bitový posun o jeden bit doleva s nasouváním logické 0.
	01	SL1	Bitový posun o jeden bit doleva s nasouváním logické 1.
	10	SLX	Bitový posun o jeden bit doleva s kopírováním LSB.
	11	ROL	Bitová rotace o jeden bit doleva.
RR	00	SR0	Bitový posun o jeden bit doprava s nasouváním logické 0.
	01	SR1	Bitový posun o jeden bit doprava s nasouváním logické 1.
	10	SRX	Bitový posun o jeden bit doprava s kopírováním MSB.
	11	ROR	Bitová rotace o jeden bit doprava.

Tabulka 3.4: Podporované operace jednotky bitových posunů a rotací



Obrázek 3.11: Blokové schéma jednotky přístupu k paměti

3.3.4 Jednotka přístupu k paměti

Jednotka přístupu k paměti (LSU, *Load/Store Unit*) zajišťuje načítání a ukládání dat z externí datové paměti. Její blokové schéma je zobrazeno na obrázku 3.11.

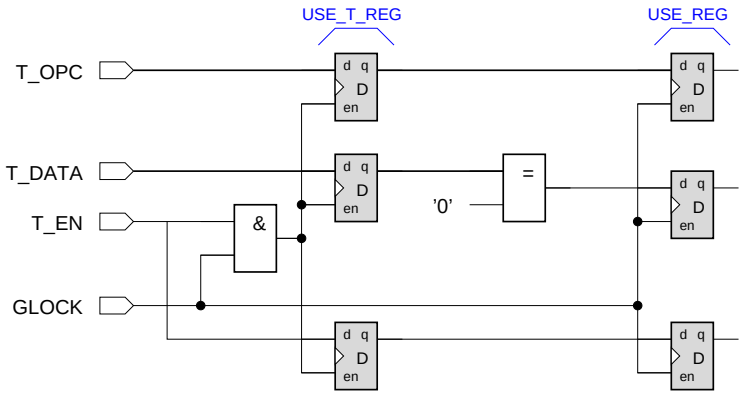
Jednotka obsahuje několik registrů. Spouštěcí registr `T_DATA` uchovává adresu a jeho operační kód `T_OPC` určuje, zda se zahájí operace načtení nebo uložení dat (viz tabulka 3.5). Povolovací vstup `T_EN` zkombinovaný s operačním kódem vytvoří řídicí signály pro čtení a zápis, které jsou zpožděny přes neřízený registr. V případě zápisu musí operandový registr `O_DATA` obsahovat platná data k uložení do paměti. V případě čtení je registrovaný řídicí signál pro čtení použit k povolení do výsledkového registru `R_DATA`. Předpokládá se, že připojená paměť má asynchronní čtení, tzn. po vystavení adresy dává na výstup požadovaná data v témže taktu. Pro podporu synchronního čtení, případně pamětí s větší latencí může být povolovací signál výsledkového registru patřičně opožděn.

3.3.5 Porovnávací jednotka

Porovnávací jednotka porovnává data na vstupy a vytváří predikáty, které se posílají do predikační jednotky. Obrázek 3.12 zobrazuje blokové schéma porovnávací jednotky na rov-

Opcode	Název	Popis
0	LOAD	Čtení dat z paměti z adresy specifikované spouštěcím registrem. Načtená data z paměti jsou uložena do výsledkového registru.
1	STORE	Uložení dat z operandového registru do paměti na adresu specifikovanou spouštěcím registrem.

Tabulka 3.5: Podporované operace jednotky přístupu k paměti



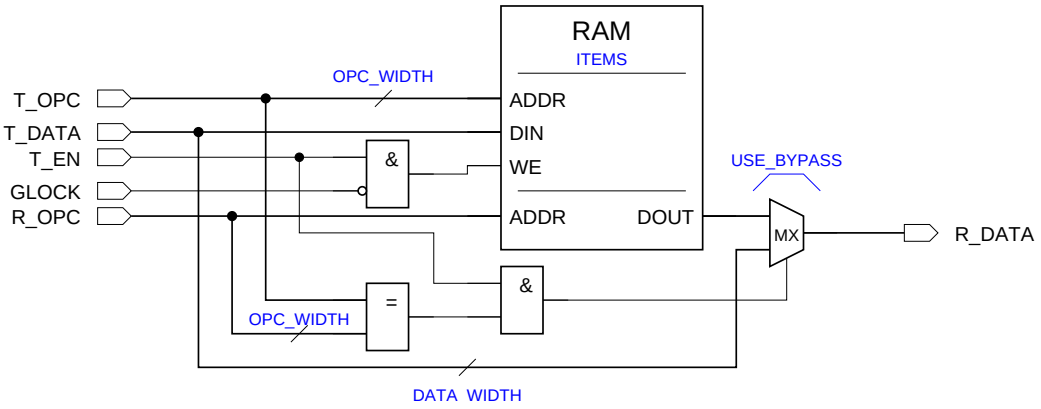
Obrázek 3.12: Blokové schéma porovnávací jednotky na rovnost nule

nost nule. Volitelné generické parametry `USE_T_REG` a `USE_REG` umožňují zapnout registry na vstupu, respektive na výstupu. Různé kombinace těchto parametrů nám umožňují uspíšit získání podmínky pro podmíněné vykonání instrukcí.

3.4 Soubory registrů

Soubory registrů jsou podobné na funkční jednotky s tím rozdílem, že provádějí operaci identity, tj. pouze ukládají data pro další použití. Obrázek 3.13 ukazuje pohled na strukturu souboru registrů implementovaného pomocí paměti RAM.

Pamět RAM má jeden čtecí a jeden zápisový port s oddělenými adresovými sběrnicemi. Datová šířka položky paměti je dána generickým parametrem `DATA_WIDTH` a jejich počet pa-



Obrázek 3.13: Struktura souboru registrů s jedním čtecím a zápisovým portem

rametrem `ITEMS`. Použitá paměť má synchronní zápis a asynchronní čtení. Aby bylo možné v jednom taktu číst a zapisovat do stejné paměťové pozice, musí být implementován prinsip předávání dat (*bypass*). Pokud se shoduje čtecí a zápisová adresa dána operačním kódem `R_OPC` a `T_OPC` a do paměti se zapisuje, dojde k předání dat ze zápisového portu přímo na výstup. Tuto možnost je možné zapnout pomocí generického parametru `USE_BYPASS`.

Kapitola 4

Realizace generického TTA procesoru GENTTA

Tato kapitola popisuje realizaci ukázkového generického TTA procesoru nazvaného GENTTA, který složen z komponent z komponent prezentovaných v předcházející kapitole. Fukčnost navrženého řešení je ověřena v simulacích. Pro ověření implementovaného prototypu procesoru v FPGA byla vybrána karta COMBO6X [8], která vznikla v rámci výzkumného projektu Liberouter. Generický procesor GENTTA je vysyntetizován pro různé parametry a porovnány vlastnosti implementace z pohledu pracovní frekvence a zabraných vzorů.

4.1 Generický procesor GENTTA

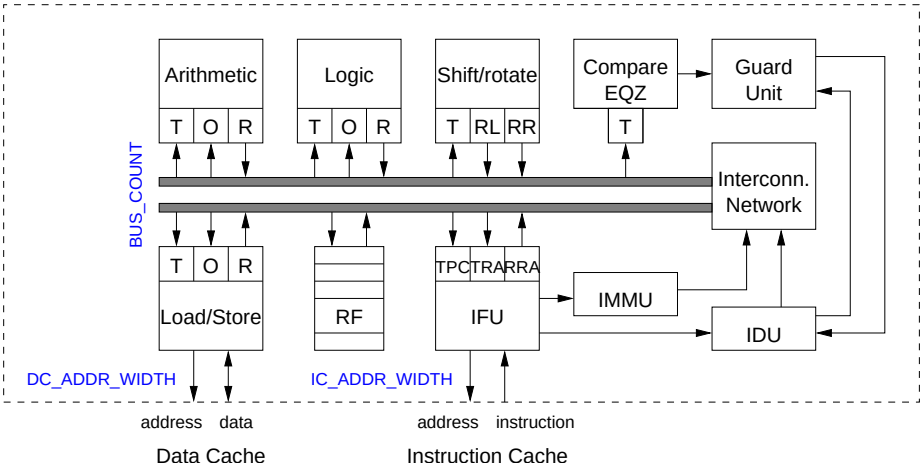
Pro ověření funkčnosti navržených a implementovaných komponent TTA architektury vznikl tento ukázkový procesor, pro ověření možnosti implementace TTA architektur na hradlovém poli FPGA. Požadavkem na ukázkový procesor je to, aby byl obecný, tzn. že na něm mohou být prováděny obecné výpočty. Mezi požadované operace a vlastnosti patří možnost větvení programu, podmíněné vykonávání kódu a podpora běžných výpočetních operací, jakou jsou aritmetické, logické, bitové posuny, atd. Pro uchovávání hodnot by měl mít k dispozici soubor registrů a možnost ukládat data do externí paměti.

Na obrázku 4.1 můžeme vidět funkční pohled na TTA procesor GENTTA. Procesor se skládá z pěti funkčních jednotek a řídicích jednotky. Mezi funkční jednotky patří aritmetická jednotka, logická jednotka, jednotka posuvů a rotací, porovnávací jednotka, jednotka přístupu k paměti a soubory registrů. Mezi řídicí jednotky patří jednotka načítání instrukcí, jednotka dekódování instrukcí, predikační jednotka a zvláštní kategorii tvoří propojovací síť typu AND-OR.

Mezi generické parametry, kterými lze ovlivnit strukturu a vlastnosti procesoru GENTTA patří:

DATA_WIDTH – tento parametr určuje datovou šířku procesoru. Je možné zadat libovolné pozitivní číslo, takže se nemusíme omezovat na procesory s 16 nebo 32 bity a můžeme mít např. procesor 13 bitový.

IC_ADDR_WIDTH – udává šířku adresové sběrnice do instrukční paměti nebo cache. Šířka instrukčního slova je dána 16-ti násobkem počtu sběrnic procesoru (na jeden přesun připadá 16 bitů – predikční identifikátor zabírá 3 bity, příznak přímého operandu jeden bit a zdrojový a cílový identifikátor každý 6 bitů).



Obrázek 4.1: Funkční pohled na generický TTA procesor GENTTA

DC_ADDR_WIDTH – udává šířku adresové sběrnice do datové paměti nebo cache.

BUS_COUNT – udává počet sběrnic procesoru, které mohou současně přenášet data. Při změně tohoto parametru se mění pouze šířka instrukčního slova.

STAGE2 – pokud je tento příznak nastaven, bude použito dvoustupňové transportní zřetězení (viz obrázek 2.6). To umožní zkrátit latenci skokových instrukcí a vyhodnocování predikátů, ale sníží pracovní frekvenci procesoru.

FU_R_REG – příznak určující, zda se má na výstupu funkčních jednotek použít výstupní registr. Jeho odstranění zkrátí latenci funkčních jednotek o jeden takt, ale opět má za následek snížení pracovní frekvence procesoru.

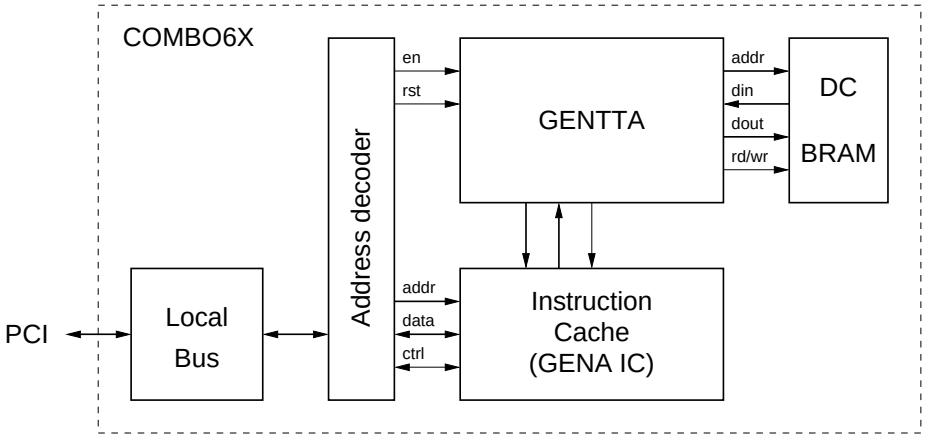
GU_BYPASS – pokud je tento generický parametr nastaven, umožňuje současný zápis a čtení příznaků do predikační jednotky. To má za následek, že latence možnosti použít predikát se zkrátí o jeden takt. Tím se opět sníží pracovní frekvence.

RU_BYPASS – tento parametr ovlivňuje možnost číst a zapisovat na jedno místo v souboru registrů v jednom taktu. Zapnutím parametru dojde ke snížení pracovní frekvence.

4.2 Realizace na kartě COMBO6X

Hardwarovou platformou pro ověření funkčnosti procesoru GENTTA je karta COMBO6X [8]. Obsahuje velké hradlové pole FPGA Virtex-II Pro XC2VP50 a malé hradlové pole Virtex-II Pro XC2VP4, které obsahuje IP core PCI nebo PCI-X. Dále obsahuje asociativní paměť CAM, statické paměti a konektor pro dynamickou paměť DDR. Karta je určena pro zapojení do klasického osobního počítače pomocí PCI sběrnice.

Pro otestování procesoru GENTTA v hardware bylo navrženo testovací zapojení na kartě COMBO6X prezentované na obrázku 4.2. Propojení s počítačem je zajištěno pomocí sběrnice PCI. Komunikaci mezi sběrnicí PCI a lokální sběrnicí zajišťuje komponenta Local Bus. K lokální sběrnici je připojen adresový dekódér, který dekóduje datové a řídící signály pro procesor GENTTA a instrukční pamětí cache. Tím je umožněno náhrát program do



Obrázek 4.2: Testovací prostředí pro ověření prototypu na kartě COMBO6X

instrukční paměti z počítače. Adresový dekodér pro procesor GENTTA generuju povolovací a resetovací signál. Procesor GENTTA dále komunikuje s datovou pamětí.

Pro provedení simulací bylo použito podpůrné simulační prostředí vytvořené v rámci projektu Liberouter. Na jednoduchém programu bylo v simulacích ověřeno, že procesor je funkční a vykonává instrukce podle programu. Po simulacích následovala syntéza a place & route do hradlové pole FPGA. Po nahrání vytvořeného firmware do karty COMBO6X byla ověřena funkčnost navrženého designu na pracovní frekvenci 100 MHz (omezení kvůli komunikací s lokální sběrnicí).

4.3 Výsledky syntézy pro FPGA Virtex-II Pro XC2VP50

Syntéza byla provedena pomocí nástroje Leonardo Precision. Pro datovou šířku procesoru 32 bitů, dvě sběrnice a výchozí nastavení ostatních parametrů (optimalizace na propustnost místo latence) dosahuje procesor GENTTA pracovní frekvence 237 MHz, při minimálním pracovním taktu 4,2 ns. Kritická cesta je tvořena řetězcem rychlých přenosů v aritmetické sčítačce. Po zřetězení této jednotky by šlo dosáhnout ještě vyšší pracovní frekvence. Zabírá 976 funkčních generátorů neboli 488 CLB buněk (cca 2% zabraných prostředků) a 540 klopných obvodů (cca 1% čipu).

Zajímavé je porovnání s implementací TTA procesoru MOVE32INT, který byl implementován pomocí CMOS technologie SoG. Technologie FPGA je obecně totiž několikanásobně pomalejší než technologie ASIC. V porovnání s pracovní frekvencí procesoru MOVE32INT, která činí 80 MHz, dosahuje procesor GENTTA trojnásobné pracovní frekvence.

Kapitola 5

Závěr

Přínos práce spočívá v navržení a implementaci generických funkčních jednotek architektury TTA. Tyto jednotky byly použity k implementaci ukázkového TTA procesoru GENTTA, který může parametrizován sadou generických parametrů. Sestavením a správným propojením navržených jednotek můžeme realizovat různé TTA procesory. Ty se mohou lišit počtem a druhem funkčních jednotek, instrukčním paralelismem daným různým počtem sběrnic a propojením funkčních jednotek.

Správnost výsledné implementace byla ověřena pomocí simulací na jednoduchém testovacím programu. Dále bylo navrženo testovací prostředí pro ověření funkčního prototypu na kartě COMBO6X. Testovací prostředí s procesorem GENTTA bylo implementováno a přeloženo pro hradlové pole Virtex-II XC2VP50. Po nahrání přeloženého firmware do FPGA byla ověřena funkčnost navrženého řešení na pracovní frekvenci 100 MHz.

Ve srovnání s implementací TTA procesoru MOVE32INT s použitím technologie CMOS SoG, která pracuje na frekvenci 80 MHz, dosahuje navržené řešení v FPGA trojnásobné pracovní frekvence 237 MHz.

Literatura

[1] Corporaal, H.: *Microprocessor architectures: from VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1998, ISBN 0-471-97157-X, 407 s.

[2] Corporaal, H.; van der Arend, P.: MOVE32INT, a sea of gates realization of a high performance transport triggered architecture. In *Proceedings of the 19th EUROMICRO Symposium on Microprocessing and Microprogramming*, Amsterdam, The Netherlands: Elsevier Science Publishers, 1993, ISSN 0165-6074, s. 53–60.

[3] Corporaal, H.; Hoogerbrugge, J.: *Code generation for Transport Triggered Architectures*, kapitola 14. Code Generation for Embedded Processors, Kluwer Academic Publishers, 1995, s. 240–259.
URL `http://citeseer.ist.psu.edu/article/corporaal95code.html`

[4] Dvořák, V.; Drábek, V.: *Architektura procesorů*. Brno: VUTIUM, 1999, ISBN 80-214-1458-8.

[5] Hoogerbrugge, J.: *Code Generation for Transport Triggered Architectures*. Dizertační práce, Delft University of Technology, Delft, The Netherlands, 1996.

[6] Hoogerbrugge, J.; Corporaal, H.: Register File Port Requirements of Transport Triggered Architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, New York, NY, USA: ACM Press, 1994, ISBN 0-89791-707-3, s. 191–195.
URL `http://ce-serv.et.tudelft.nl/MOVE/papers/micro27.ps.gz`

[7] Janssen, J.; Corporaal, H.: Partitioned Register File for TTAs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, ISBN 0-8186-7349-4, s. 303–312.
URL `http://ce-serv.et.tudelft.nl/MOVE/papers/micro28.ps.gz`

[8] Liberouter, P.: Karta COMBO6X.
URL `http://www.liberouter.org/card_combo6x.php`

[9] Wikipedia: Pentium — Wikipedia, The Free Encyclopedia. 2007.
URL `http://en.wikipedia.org/wiki/Pentium`

[10] Wikipedia: TMS320 — Wikipedia, The Free Encyclopedia. 2007.
URL `http://en.wikipedia.org/wiki/TMS320`

[11] Wikipedia: TriMedia — Wikipedia, The Free Encyclopedia. 2007.
URL `http://en.wikipedia.org/wiki/TriMedia`

[12] Xilinx: *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.* 2007.
URL `http://www.xilinx.com/bvdocs/userguides/ug012.pdf`

Seznam zkratek a symbolů

CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GPR	General-Purpose Register
ILP	Instruction Level Paralelism
LSB	Least Significant Bit
MSB	Most Significant Bit
OTA	Operation Triggered Architecture
RISC	Reduced Instruction Set Computer
SVTL	Semi Virtual-Time Latching
TTA	Transport Triggered Architecture
TVTL	True Virtual-Time Latching
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word (architektura nebo procesor)
VTL	Virtual-Time Latching

Seznam příloh

Příloha A

Popis rozhraní implementovaných jednotek

Generické parametry			
BUS_COUNT	positive	-	Počet sběrnic.
SRC_COUNT	positive	-	Počet zdrojových zásuvek.
SRC_ID_COUNT	positive	-	Šířka zdrojového identifikátoru.
DST_COUNT	positive	-	Počet cílových zásuvek.
DST_ID_COUNT	positive	-	Šířka cílového identifikátoru.
USE_REG	boolean	true	Zapíná výstupní registry dekódovaných signálů.
Porty			
CLK	std_logic	in	Hodinový signál.
RESET	std_logic	in	Resetovací signál.
GLOCK	std_logic	in	Globální blokování.
Rozhraní s jednotkou načítání instrukcí			
SRC_ID	slv(BUS_COUNT)[SRC_ID_WIDTH]	in	Zdrojové identifikátory.
DST_ID	slv(BUS_COUNT)[DST_ID_WIDTH]	in	Cílové identifikátory.
Rozhraní s predikační jednotkou			
SQUASH	std_logic(BUS_COUNT)	in	Potlačovací signál.
Signály zdrojových zásuvek			
SRC_CONST	slv(SRC_COUNT)[SRC_ID_WIDTH]	in	Konstantní adresy zásuvek
SRC_SEL	slv(SRC_COUNT)[BUS_COUNT]	out	Výběrový signál (kódovaný one-hot).
SRC_OPC	slv(SRC_COUNT)[SRC_ID_WIDTH]	out	Operační kód.
SRC_EN	std_logic(SRC_COUNT)	out	Povolovací signál.
Signály cílových zásuvek			
DST_CONST	slv(SRC_COUNT)[DST_ID_WIDTH]	in	Konstantní adresy zásuvek.
DST_SEL	slv(DST_COUNT)[BUS_COUNT]	out	Výběrový signál (kódovaný binárně).
DST_OPC	slv(DST_COUNT)[DST_ID_WIDTH]	out	Operační kód.
DST_EN	std_logic(DST_COUNT)	out	Povolovací signál.

Tabulka A.1: Rozhraní jednotky dekódování instrukcí

Generické parametry			
DATA_WIDTH	positive	-	Datová šířka.
BUS_COUNT	positive	-	Počet sběrnic.
SRC_ID_WIDTH	positive	-	Šířka zdrojového identifikátoru.
USE_REG	boolean	true	Zapne výstupní registry.
SGN_EXT	boolean	true	Povolí znaménkové rozšíření.
Porty			
CLK	std_logic	in	Hodinový signál.
RESET	std_logic	in	Resetovací signál.
GLOCK	std_logic	in	Globální blokování.
SRC_ID	slv(BUS_COUNT)[SRC_ID_WIDTH]	in	Zdrojové identifikátory.
IMM_DATA	slv(BUS_COUNT)[DATA_WIDTH]	out	Přímé operandy.

Tabulka A.2: Rozhraní jednotky přímých operandů

Generické parametry			
DATA_WIDTH	positive	-	Datová šířka.
BUS_COUNT	positive	-	Počet sběrnic.
SRC_COUNT	positive	-	Počet zdrojových zásuvek.
DST_COUNT	positive	-	Počet cílových zásuvek.
Porty			
Přímé operandy			
IMM_DATA	slv(BUS_COUNT)[DATA_WIDTH]	in	Data.
IMM_SEL	std_logic(BUS_COUNT)	in	Povolovací signál.
Zdrojové zásuvky			
SRC_DATA	slv(SRC_COUNT)[DATA_WIDTH]	in	Data.
SRC_SEL	slv(SRC_COUNT)[BUS_COUNT]	in	Výběrový signál (kódovaný one-hot). Udává na kterou sběrnici se mají vystavit data odpovídající zdrojové zásuvky.
Cílové zásuvky			
DST_DATA	slv(DST_COUNT)[DATA_WIDTH]	out	Data.
DST_SEL	slv(DST_COUNT)[log2p(BUS_COUNT)]	in	Výběrový signál (kódovaný binárně). Vybírá z které sběrnice se pošlou data do odpovídající cílové zásuvky.

Tabulka A.3: Rozhraní propojovací sítě

Generické parametry			
DATA_WIDTH	positive	-	Datová šířka.
Porty			
CLK	std_logic	in	Hodinový signál.
RESET	std_logic	in	Resetovací signál.
GLOCK	std_logic	in	Globální blokování.
Operandový registr			
O_DATA	slv[DATA_WIDTH]	in	Data.
O_EN	std_logic	in	Povolovací signál.
Spouštěcí registr			
T_DATA	slv[DATA_WIDTH]	in	Data.
T_OPC	slv[1]	in	Operační znak.
T_EN	std_logic	in	Povolovací signál.
Výsledkový registr			
R_DATA	slv[DATA_WIDTH]	in	Data.

Tabulka A.4: Rozhraní aritmetické jednotky

Generické parametry			
DATA_WIDTH	positive	-	Datová šířka.
Porty			
CLK	std_logic	in	Hodinový signál.
RESET	std_logic	in	Resetovací signál.
GLOCK	std_logic	in	Globální blokování.
Operandový registr			
O_DATA	slv[DATA_WIDTH]	in	Data.
O_EN	std_logic	in	Povolovací signál.
Spouštěcí registr			
T_DATA	slv[DATA_WIDTH]	in	Data.
T_OPC	slv[2]	in	Operační znak.
T_EN	std_logic	in	Povolovací signál.
Výsledkový registr			
R_DATA	slv[DATA_WIDTH]	in	Data.

Tabulka A.5: Rozhraní logické jednotky

Generické parametry			
DATA_WIDTH	positive	-	Datová šířka.
Porty			
CLK	std_logic	in	Hodinový signál.
RESET	std_logic	in	Resetovací signál.
GLOCK	std_logic	in	Globální blokování.
Spouštěcí registr			
T_DATA	slv[DATA_WIDTH]	in	Data.
T_OPC	slv[2]	in	Operační znak.
T_EN	std_logic	in	Povolovací signál.
Výsledkové registry			
R_L_DATA	slv[DATA_WIDTH]	in	Výsledek posuvů a rotací doleva.
R_R_DATA	slv[DATA_WIDTH]	in	Výsledek posuvů a rotací doprava.

Tabulka A.6: Rozhraní jednotky bitových posunů a rotací