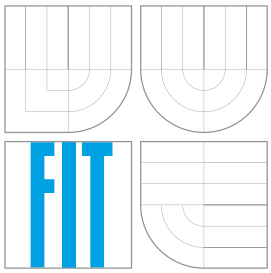
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODEL PROCESORU NIOS II

NIOS II PROCESSOR MODEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK MASAŘÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Masařík Marek**
Obor: Informační technologie
Téma: **Model procesoru NIOS II**
NIOS II Processor Model
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s instrukční sadou procesoru NIOS II.
2. Seznamte se s dostupnými dokumenty popisujícími architekturu procesoru NIOS II.
3. Seznamte se s jazykem CodAL pro modelování procesorových architektur a systémů na čipu.
4. Vyberte podmnožinu vhodných instrukcí a vytvořte instrukční model procesoru NIOS II v jazyce CodAL.
5. Dle dostupné dokumentace vytvořte obvodový model procesoru NIOS II v jazyce CodAL.
6. Na vhodné sadě testovacích aplikací ověřte funkčnost vytvořeného modelu.
7. Zhodnoťte vytvořené řešení a dosažené výsledky.

Literatura:

- Altera. *NIOS II Gen2 Processor Reference Guide, NII5V1GEN2.* Altera Corporation, 2015.
- Codasip. *Codasip Studio User Guide.* Codasip s.r.o., 2015

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 4 zadání.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Šimková Marcela, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Cílem této bakalářské práce bylo vytvoření návrhu modelu procesoru Nios II v jazyce pro popis architektur procesoru zvaném CodAL. Návrh procesoru probíhal na dvou úrovních abstrakce. První úroveň se skládala z popisu instrukční sady a druhá z návrhu architektury a implementace hardwarového modelu. Důležitou součástí návrhu procesoru je testování a verifikace, které proběhly úspěšně na připravené benchmarkové testovací sadě. Výsledný procesor je tak možné potenciálně využít v reálných aplikacích.

Abstract

This bachelor thesis deals with the implementations of Nios II processor model in the description language processor called description CodAL. The implementation of processor is on two levels of abstraction. First level of abstraction is the instruction accurate model and second is the cycle accurate model. An important part of processor design is testing and verification which were realized on the prepared benchmark set. The resulting processor can be potentially used in real applications.

Klíčová slova

Procesor, Nios II, ASIP, RISC, instrukční sada, CodAL, Codasip, modelování, simulace, zřetězená linka

Keywords

Processor, Nios II, ASIP, RISC, instruction set, CodAL, Codasip, modeling, simulation, pipeline

Citace

Marek Masařík: Model procesoru NIOS II, bakalářská práce, Brno, FIT VUT v Brně, 2016

Model procesoru NIOS II

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Marcely Zachariášové, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Masařík
18. května 2016

Poděkování

Chtěl bych poděkovat vedoucí své bakalářské práce Ing. Marcele Zachariášové, Ph.D. za cenné rady při vypracování práce. Poděkování také patří Hynku Bláhovi a Radku Hájkovi za cenné rady při návrhu a realizaci modelu.

Obsah

Kapitola 1

Úvod

V dnešní době se počítačové systémy nacházejí téměř kdekoliv kolem nás a jsou součástí našeho každodenního života. Příkladem jsou vestavěné systémy schopné se připojit k internetu, tzv. internet věcí (*Internet of Things*) nebo počítače nové generace. Jejich úspěšnost je spojena s rychle rostoucím vývojem aplikačně specifikovaných procesorů zaměřených na výkon, úsporu energie a velikost jádra procesoru na čipu. Softwarové modelování procesoru umožňuje snadněji navrhnout a díky vysoké abstrakci jednoduše testovat a následně optimalizovat.

Touto problematikou se zabývá firma Codasip s.r.o, která vytvořila vývojové prostředí zvané Codasip studio pro modelování aplikačně specifikovaných procesorů (*ASIP*), generující automaticky vývojové a ladící nástroje. Dále vyvinula a rozvíjí jazyk pro popis procesoru CodAL. Základní informace o jazyce CodAL a vývojovém prostředí jsou popsány v kapitole 3.

Vývojové prostředí Codasip Studio a jazyk CodAL jsem využil při tvorbě návrhu procesoru Nios II, které umožňuje snadnou úpravu modelu a doplnění dalších instrukcí do návrhu procesoru. Narozdíl od stávajícího řešení implementace procesoru Nios II firmou Altera, které zahrnuje jednu instrukční sadu a 3 fixní typy architektur, se liší rychlostí zpracovaní instrukcí a plochou FPGA. Tím je omezena individuální úprava modelu podle vlastních požadavků, kterých lze dosáhnout jazykem CodAL.

V kapitole 2 jsou popsány teoretické informace o procesorech, jejich dělení, zpracování instrukcí ve zřetězené lince a jazyky, které se používají pro tvorbu návrhu instrukčního nebo hardwarového popisu procesoru.

Dále se bakalářská práce zaměřuje na popis procesoru Nios II a instrukční sady. Implementací instrukčního modelu se zabývá kapitola 5, který zahrnuje i zapojení jádra na platformě. Předposlední kapitola je určena implementaci hardwarovému modelu.

Testování a verifikace je nedílnou součástí vývoje každého návrhu procesoru, které zahrnuje ověření implementace instrukční sady, správného provádění a odhalení chyb. Kapitola 7 se také zabývá syntézou procesoru a nahráním na FPGA.

Kapitola 2

Teorie

Tato kapitola se zabývá problematikou mikroprocesorů, jejich zařazením, využitím zřetězeného zpracování a jazyků pro popis architektury mikroprocesorů. Informace k této kapitole jsem čerpal ze zdroje [7] a [5].

2.1 Mikroprocesor

Mikroprocesor je víceúčelové programovatelné zařízení, které provádí sled operací (instrukcí) uložených v paměti, a tím je provedena určitá funkce. Samotný mikroprocesor označujeme jako řídící jednotku (jádro), který ke svému provozu potřebuje vstupní a výstupní periferie.

Procesory dělíme podle několika základních kritérií:

Podle instrukční sady:

- CISC (*Complex Instruction Set Computer*) – komplexní instrukční sada s velkým množstvím instrukcí. Obsahuje instrukce vykonávající speciální funkce, které efektivněji využívají paměť, ale potřebují více času při jejich zpracování, to vede k větší složitosti hardwaru.
- RISC (*Reduced Instruction Set Computer*) – redukovaná instrukční sada, využívající menší počet složitých instrukcí. Tyto instrukce jsou realizovány pomocí obecnějších instrukcí. Instrukce mají pevnou bitovou délku a jeden nebo menší počet formátů instrukcí. Operace nad pamětí se provádí pomocí dvou instrukcí Load a Store.

Podle architektury procesoru:

- Harvardská architektura – fyzicky odděluje adresový prostor pro paměť programu a data.
- Von Neumannova architektura – je charakteristická tím, že program i data se nachází ve stejném paměťovém prostoru.

Podle šířky slova:

Je nejmenší počet bitů, se kterými je schopen mikroprocesor pracovat v rámci jedné operace. Jedná se o šířku vnitřní datové sběrnice CPU. Nejčastější šířky slova jsou 8,16,32 nebo 64 bitů.

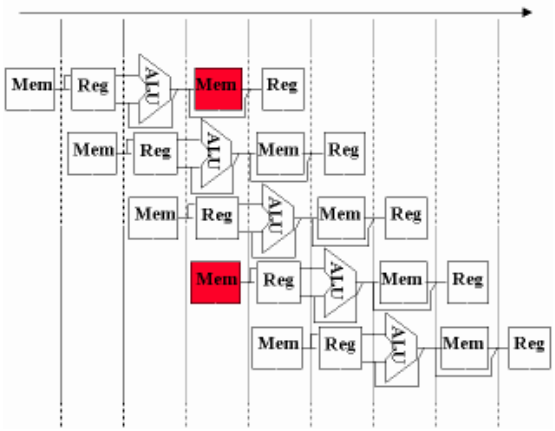
2.2 Zřetězené zpracování

Zřetězené zpracovaní (*ang. pipeling*) je implementační technika, která využívá paralelismu ke zpracování více instrukcí ve stejnou chvíli. Tato technika zajišťuje zrychlení systému. Zřetězená linka se skládá z několika kroků, kdy každý z nich provádí jednotlivé operace paralelně s ostatními kroky. Každý krok se nazývá fáze linky (*ang. pipe stage*).

Hazard

Je to situace, kdy ve zřetězené lince nemůže být provedena další instrukce v následujícím hodinovém taktu. To může vést k nesprávnému provedení výpočtu. Rozlišujeme tři základní typy hazardu: strukturální, datový a řídící hazard.

1. **Strukturální hazard** – k tomuto hazardu dochází, když dvě instrukce potřebují ve stejnou dobu hardwarové zdroje (paměť, ALU, periferie), přičemž nepodporují vícenásobný přístup (obrázek 2.1). V případě, že sled instrukcí narazí na tento hazard, instrukce bude v lince pozastavena do doby, než budou k dispozici požadované zdroje.



Obrázek 2.1: První instrukce zapisující výsledek operace do paměti a snahu načíst ve stejnou chvíli další instrukci z paměti.

2. **Datový hazard** – vzniká, pokud některá instrukce v lince čeká na dokončení jiné instrukce, která ještě nebyla dokončena. Pokud se v lince nachází dvě instrukce operující nad stejnými daty, rozlišujeme tři případy datových hazardů:

   a) Čtení po zápise (*Read After Write – RAW*) – první instrukce uloží vypočtená data do paměti a následující instrukce pracuje se stejnými daty. Pokud následující instrukce čte data dříve, než byla dokončena předchozí instrukce, jsou přečtená data neplatná.

   b) Zápis po čtení (*Write After Read – WAR*) – první instrukce čte data ze stejné adresy, do které následující instrukce zapisuje. Pokud zápis proběhl před čtením, první instrukce pracuje s neplatnými daty.

   c) Zápis po zápise (*Write After Write – WAW*) – nastává, pokud dvě instrukce zapisují data na stejnou adresu. Pokud druhá instrukce provede zápis dříve než první, objeví se v paměti po dokončení instrukcí neplatná data.

3. **Řídící hazard** – vzniká při provádění skokových instrukcí. Pokud není možné při načtení skokové instrukce do linky určit, jestli bude skok na adresu proveden, může nastat řídící hazard. Zjištění okolností případného skoku v pozdějších částech linky má za následek načtení neplatných instrukcí do linky. Je nutné tyto instrukce vyjmout a nahradit instrukcemi z adresy skoku. Další možným řešením je tzv. predikce skoku, která již při načtení instrukce zjišťuje, zda-li bude skok na požadovanou adresu proveden. Predikci skoku dělíme na statickou a dynamickou, přičemž statická predikce se váže k určitým signálům zřetězené linky.

2.3 Jazyky pro popis architektury

Jazyky pro popis hardwaru (*Hardware Description Language - HDL*) slouží pro modelování a simulaci pouze hardwaru. Tyto jazyky mají mnoho nedostatků, například, že model specifikovaný v HDL neobsahuje některé informace např. syntaxe assembleru. Pro účel popsání modelu mikroprocesoru včetně dodatečných informací týkajících se softwarových nástrojů byl vyvinut jazyk pro popis architektur (Architecture Description Languages - ADL). [6]

Jazyky ADL rozdělujeme do následujících kategorií:

- jazyky pro popis struktury,
- jazyky pro popis instrukční sady a
- smíšené jazyky.

Jazyky pro popis struktury

Důležitým prvkem při návrhu ADL je nalézt kompromis mezi obecností a abstrakcí. Díky velkému množství typů architektur je obtížné nalézt vysoký stupeň formalismu pro popis základních vlastností všech typů mikroprocesoru. Toho je docíleno pomocí nižšího stupně abstrakce a získání detailnějšího popisu architektury. Do této skupiny patří například jazyky: MIMOLA, AIDL.

Jazyky pro popis instrukční sady

Tyto jazyky se zaměřují pouze popisem instrukční sady a ne strukturou a implementací mikroarchitektur. Sémantika instrukcí je přímo určena ve formě meziregistrových přenosů a detaily hardwarové struktury jsou ignorovány. Tyto jazyky jsou prioritně vyvíjeny pro vytvoření přenositelného překladače. Jazyky pro popis instrukční sady jsou např. jazyky ISDL, nMl a CSDL.

Smíšené jazyky

Smíšené spojují strukturální jazyky a jazyky pro popis instrukční sady. Snaží se vyhnout nedostatku informací pro simulaci a zároveň zachovat určitě množství, aby se nezpomalovala simulace a generování softwarových nástrojů. Smíšené jazyky obsahují prvky obou typů jazyků. Patří sem jazyky např. LISA, EXPRESSION, FlexWare, nebo CodAL, kterému je věnována následující kapitola.

Kapitola 3

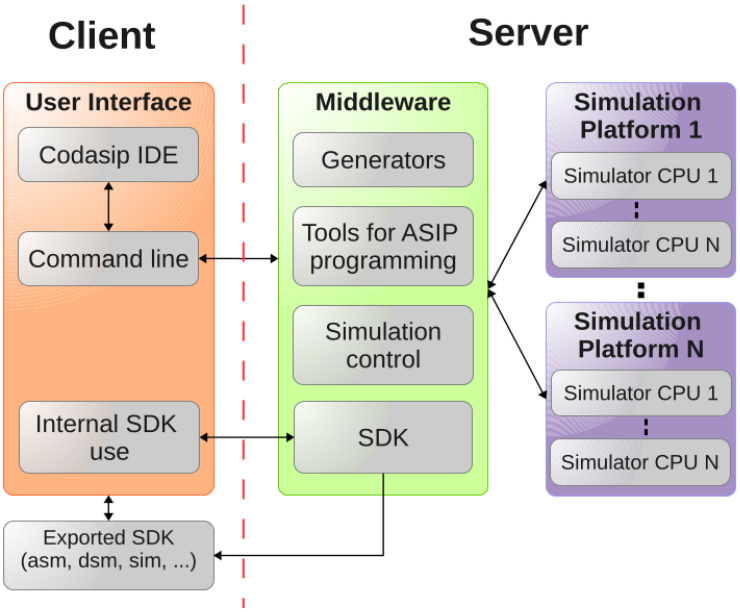
Vývojové prostředky

V této kapitole je popsány jednotlivé části vývojového prostředí Codasip Studia. [3]

Codasip Studio jsou automatizované, integrované vývojové prostředí založené na Eclipse, které umožňuje vytvořit aplikačně specifikované procesory (*ASIP – Application-Specific Instruction set Processor*) a multiprocesorové systémy na čipu (*MPSoC – Multiprocessor Systems on Chip*).

Návrh procesoru začíná na vysoké úrovni abstrakce v jazyce CodAL. To umožňuje rychlejší návrh ASIP procesoru a automatizace úloh, které by jinak musely být vykonávány manuálně. Další popis jazyka CodAL je v podkapitole 3.2. Codasip Studio umožňuje návrháři automaticky generovat simulační nástroje, prototypy virtuálních systémů a verifikační prostředí.

Vývojové prostředí pracuje na třech úrovních: uživatelské rozhraní, server a simulační vrstva, jak je zobrazeno na obrázku 3.1. Tyto vrstvy spolu komunikují pomocí TCP/IP protokolu.



Obrázek 3.1: Úrovně vývojového prostředí Codasip studia

Uživatelské rozhraní přijímá vstupy od vývojářů (např. příkaz k zahájení simulace) a zobrazuje informace o jejich průběhu. Návrh procesoru lze provádět pomocí grafického uživatelského rozhraní na bázi programu Eclipse nebo příkazových řádků, které umožňují skriptování nebo automatické testování.

Server (neboli middleware) přijímá příkazy z uživatelského rozhraní a ty zpracovává a předává zpět. Server je také zodpovědný za instalování simulátoru do simulační vrstvy.

3.1 Software Development Kits

Software Development Kits (*SDK*) je sada nástrojů ve vývojovém prostředí pro podporu automatizovaného vývoje ASIP procesoru. Mezi tyto nástroje patří:

Assembler

Assembler je nástroj, který převádí jazyk symbolických instrukcí do objektového souboru. Objektové soubory jsou poté spojeny linkerem do binárního kódu čitelného procesorem. Codasip Assembler je schopen zpracovávat dva jazyky najednou. První je definován syntaxí jednotlivých instrukcí procesoru, které jsou součástí popisu jazyka CodAL. Druhý jazyk se používá k určení direktivy a symbolu jazyka symbolických instrukcí.

Disassembler

Disassembler je nástroj, který zpracovává spustitelný soubor a převádí ho zpět na původní kód jazyka symbolických instrukcí.

Linker

Úkolem tohoto nástroje je spojit více objektových souborů vygenerovaných assemblerem do jednoho spustitelného souboru.

Jazyk překladače C/C++

Tento nástroj je založen na populárním a široce používaném open source LLVM Frameworku. V kompilátoru je možnost nastavit různá omezení, to je výhodou například při vzniku instrukčních hazardů a při jejich eliminaci.

Simulátor

Simulátor umožňuje návrhářům ladit a testovat vytvořené hardwarové a softwarové systémy. Všechny typy simulátoru jsou generovány z platformy CodAL modelu. Podle abstrakce modelu rozlišujeme tři typy simulátoru: instrukční sady, hardwarové složky a QEMU simulátor.

První a druhý typ simulátoru jsou založeny na podobném principu sledování stavu modelu (paměti a registru) po načtení, dekódování a provedení instrukce. Rozdíl mezi nimi je krok simulace, první je prováděn po instrukci a druhý po hodinovém taktu procesoru.

QEMU je velmi rychlý instrukčně přesný simulátor (také zvaný emulátor), který může simulovat více než 1000 MIPS[1] na standardním počítači. Proto je vhodný pro dlouhodobé provádění testů, kdy výkon instrukčně přesného simulátoru není dostatečný.

[1] MIPS – milion instrukcí za sekundu.

Debugger

Tento nástroj může být generovaný společně se simulátorem. To umožňuje ladit zdrojové kódy přímo v modelu procesoru pomocí jeho simulátoru. Debugger je založen na open source standardní ladící platformě GDB (*GNU Project Debugger*) a podporuje základní funkce jako je umísťování breakpointů, watchpointů nebo krokování.

Profiler

Funkční simulátor není často dostatečný pro vytvoření optimálního návrhu. K tomu je třeba podrobnější informace o simulačním procesu. Profiler zaznamenává všechny důležité informace o průběhu simulace, například kolik hodinových cyklů konkrétní funkce trvá nebo využití instrukce během simulace. Výsledky lze zobrazit v podobě seznamu instrukcí nebo grafů. Na základě těchto statistik lze rozhodnout, které části procesoru je vhodné optimalizovat.

3.2 Jazyk CodAL

Tato podkapitola vychází ze zdroje [2]

Jazyk CodAL řadíme mezi jazyky pro popis architektur. Využívá se pro souběžný návrh hardwaru a softwaru (*Hardware/Software Codesign*) víceprocesorových systémů integrovaných na jeden čip (*MPSoC*). Z vytvořeného modelu pomocí jazyka CodAL lze automaticky generovat programovací a simulační nástroje. Ze stejného modelu jde také automaticky vytvořit implementaci mikroarchitektury v hardwaru, a to pomocí VHDL nebo Verilog jazyků. Popis procesoru je rozdělen na dvě části: platforma a samotné jádro procesoru.

Popis platformy je zaměřen na popis prostředí ASIP. To může obsahovat definici vnějších zdrojů, kterými jsou paměť, komponenty, sběrnice a jejich propojení.

Popis jádra procesoru je hlavní částí modelu procesoru a je oddělena od popisu platformy. Popis jádra se skládá ze čtyř částí:

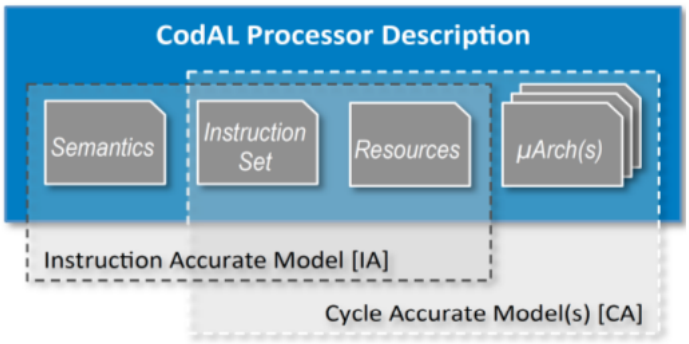
- zdroje architektury – programový čítač, registry,
- instrukční sada – názvy instrukcí, jejich binární forma (operační kódy),
- sémantika – chování každé instrukce, výjimky,
- realizace – zdroje a chování (časování), které definují konkrétní mikroarchitekturu.

Rozlišujeme dva druhy ASIP modelu:

- Architektonický model neboli model na úrovni instrukční sady (*Instruction Accurate Model – IA model*)
- Mikro-architekturní model neboli model na úrovni cyklů (*Cycle Accurate Model – CA model*)

Model na úrovni instrukční sady využívá zdroje, instrukční sadu a její sémantickou část. Model na úrovni cyklů využívá zdroje, instrukční sadu a implementační čas (znázorněno na obrázku 3.2).

Model procesoru popsaný v jazyce CodAL je převeden do interní reprezentace ve formátu XML. Načtený XML popis se použije pro automatické generování nástrojů assembleru, disasembleru, linkeru, překladače jazyka C a simulátorů popsaných v kapitole 3.1.



Obrázek 3.2: CodAL popis procesoru

3.3 Struktura jazyka

Instrukční sada

Každá instrukční sada je definována pomocí dvou základních konstrukcí *element* a *set*. Nejprve jsou implementovány jednotlivé elementy, které se sdružují do kolekcí pomocí *set*u a umožňují vytvářet složitější elementy.

```
element jméno
{
    use gpr as reg;            // použití elementu
    assembler{ "add" reg }     // instrukce zapsaná v assembleru
    binary{ 0x00:bit[6] reg } // binárně zakódovaná instrukce
    semantics{...}             // definuje chování elementu
    return{...}                // definuje návratovou hodnota
    timing{...}                // definuje aktivování událostí
}
```

Zdrojový kód 3.1: Struktura elementu jazyka CodAL

Hardwarová implementace

Model na úrovni cyklů se skládá z jednotlivých událostí (`event`). Událost má podobnou strukturu jak element a může aktivovat jiné události a dekodéry (`decoder`). Dále definuje sekci `timing`, která definuje, v jakém pořadí budou události v modelu aktivovány. Sekce `decoder` popisuje instrukční dekodér.

Sémantika

Sekce `sematics` popisuje chování jednotlivých elementů nebo událostí. K tomuto popisu je použit jazyk ANSIC C, který specifikuje chování jednotlivých instrukcí. Jazyk využívá všechny operandy jazyka C až na ukazatele. Proto parametry jsou předávány pouze hodnotou ve standardním datovém typu.

Kapitola 4

NIOS II

Procesor NIOS II od společnosti Altera je 32bitový procesor založený na RISC architektuře. Je konfigurovatelný a obsahuje jádro typu soft-core. Samotné jádro procesoru používá malý počet logických elementů obvodu FPGA, to umožnuje využít další komponenty nebo připojit vlastní hardware. Je tak možné konfigurovat komponenty i instrukce. [1]

Procesor může být realizován ve třech různých konfiguracích, aby splňoval požadavky systému:

- Nios II/f je "rychlý"
- Nios II/s je "standard"
- Nios II/e je "ekonomický"

Nios II/f je „rychlý“ – verze určená pro vyšší výkon. Má širší rozsah konfigurace, což umožňuje optimalizovat procesor pro výkon, ale na úkor velikosti.
Hlavní rysy jádra:

- Jednotka pro správu paměti (MMU)
- Jednotka pro ochranu paměti (MPU)
- 2 GB externího adresového prostoru bez MMU, jinak až 4 GB
- Rozšířená podpora přerušení
- Konektor externího vektoru přerušení
- JTAG ladicí modul
- Obsahuje instrukční i datovou cache[1]
- 6-ti stupňová zřetězená linka
- Dynamická predikce skoku
- Volitelné hardwarové násobičky, dělení a posunu
- Podpora až 256 vlastních instrukcí

[1] Cache – vyrovnávací paměť

Nios II/s je „standard“ – tato verze je kompromis mezi velikostí FPGA a výkonem procesoru.
Hlavní rysy jádra:

- Podpora instrukční cache, ale ne datové
- Až 2 GB externího adresového prostoru
- 5-ti stupňová zřetězená linka
- Statická predikce skoku
- Volitelné hardwarové násobičky, děličky a posunu
- Podpora vlastních instrukcí
- JTAG ladicí modul

Nios II/e je „ekonomický“ – verze, která vyžaduje nejmíň zdrojů FPGA.
Hlavní rysy jádra:

- Vykonání jedné instrukce během 6 hodinových taktů
- Až 2 GB externího adresového prostoru
- JTAG ladící modul
- Nepodporuje predikci skoku
- Kompletní jádro v méně než 700 logických elementech
- jednostupňová zřetězená linka

4.1 Adresování

NIOS II je 32bitový procesor. Do tohoto adresového prostoru se může číst a zapisovat po slovech (32 bitů), půlslovech (16 bitů), nebo po bajtech (8 bitů) dat.

Rozlišujeme pět adresovacích režimů:

Registrové adresování – instrukce obsahuje na 5 bitech adresu jednoho registru z registrového pole.

Nepřímé adresování dat – adresa datového prostoru je dána součtem registru a instrukčního 16bitového znaménkového operandu. Tuto metodu adresování využívají instrukce `ldw` a `stw`.

Přímé adresování programové paměti – cílová adresa je součástí 26-bitového operandu instrukce, která je v posunuta o 2 bity doleva a spojena s horními 4-bity programového čítače. Toto adresování se využívá u instrukcí `call` a `jmpi`, kde $PC_{31..28}$ definuje rozsah provedení skoků.

Nepřímé adresování programové paměti – toto adresování se využívá při realizaci nepodmíněných skoků nebo volání programu `callr` a `jmp`. Cílová adresa je určena obsahem registru instrukce.

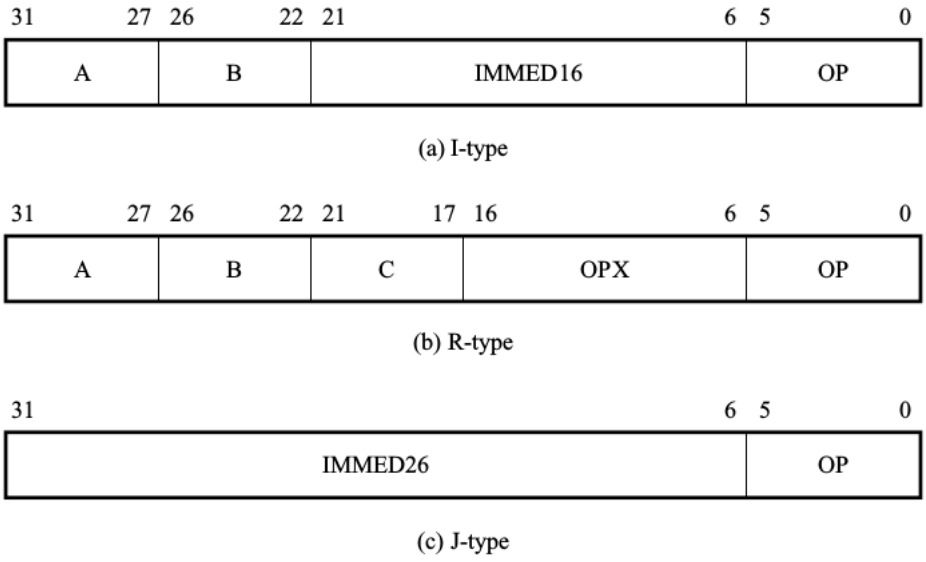
Relativní adresování – využívají instrukce pro relativní podmíněné skoky. Adresa skoku je určena jako součet adresy následující instrukce (*PC+1*) se znaménkovou konstantou nacházející se na 16 bitech v operačním kódu instrukce. Jedná se tedy o relativní podmíněný skok o hodnotu konstanty.

4.2 Instrukční sada

Všechny instrukce NIOS II mají 32bitovou šířku. Přehled všech instrukcí je dostupný v dokumentaci od Altery [1] v kapitole Instruction Set Reference. Další typ instrukcí tvoří pseudoinstrukce, kde assembler nahradí každou pseudoinstrukci jednou nebo více binárními částmi z jiných instrukcí.

Obrázek 4.1 zobrazuje tři možné formáty instrukcí: I-type, R-type a J-type. Ve všech případech je na spodních šesti bitech operační kód (OP) pro rozlišení instrukcí. Zbývající bity se používají k specifikaci registru, přímé hodnoty nebo prodloužení operačního kódu (OPX).

- I-type – 5bitové pole A a B se používá k určení registru z registrového pole. A 16bitové pole IMMED16 je přímá hodnota.
- R-type – A, B a C specifikuji použité registry a 11bitový OPX je rozšířením operačního kódu.
- J-type – 26bitový IMMED26 obsahuje neznamínkovou přímou hodnotu, která se používá v instrukci volání.



Obrázek 4.1: Formát instrukcí procesoru NIOS II

4.3 Typy instrukcí

Instrukční sadu je možné rozdělit do několika typů instrukcí:

- Instrukce nad pamětí (Load-store)
- Aritmetické a logické instrukce
- Instrukce přesunu
- Instrukce porovnání
- Instrukce posunu a rotace
- Skokové instrukce

Instrukce nad pamětí

Instrukce nad pamětí Load a Store slouží k přesouvání 32 bitů dat mezi registry, pamětí a periferiemi. Dále jsou to instrukce formátu I-typ, například instrukce LDW (*Load Word instruction*), která pomocí efektivní adresy určuje pozici čtených dat z paměti. Je dána součtem obsahu zdrojového registru a ofsetu obsažená v instrukci. Ofset je 16-bitová hodnota znaménkově roztažena na 32 bitů. Operand načtený z paměti o šířce 32 bitů je uložen do cílového registru.

Dále instrukce STW (*Store Word instrucion*) pracuje obdobně jako LDW, namísto načítání ukládá obsah zdrojového registru do paměti na adresu danou součtem registru a ofsetu. Instrukce Load a Store dále používají operandy o bitové šířce 8 a 16 bitů. Mezi tyto instrukce patří:

- ldb – načítání 8 bitů, kterou budou znaménkově roztaženy do cílového registru
- ldbu –načtení 8 bitů do cílového registru
- ldh – načítání 16 bitů, kterou budou znaménkově roztaženy do cílového registru
- ldhu – načtení 16 bitů do cílového registru
- stb – ukládá do paměti spodních 8 bitů zdrojového registru
- sth – ukládá do paměti spodních 16 bitů zdrojového registru

Aritmetické a logické instrukce

Aritmetické instrukce provádějí operace s daty dané dvěma vstupními registry a výsledek je uložen do cílového registru nebo jedním vstupním registrem a 16-bitovou konstantou specifikovanou v instrukci. Například instrukce sčítání využívá tuto 16-bitovou konstantu jako znaménkový a bezznaménkový operand, jelikož neobsahuje žádný příznak, který by jej specifikoval. To znamená, že při použití bezznaménkového operandu při sčítání je detekován nejvýznamnější bit a oddělen od operandu. Tyto instrukce jsou formátu R-typ nebo I-typ a patří do této skupiny:

- add – součet registrů
- addi – součet registru a konstanty

- sub – odčítání registrů
- subi – odčítání konstanty od registru
- mul – násobení registrů
- muli – násobení registru a konstanty
- div – dělení
- divu – bezznaménkové dělení

Logické instrukce jsou prováděny AND, OR, XOR a NOR operacemi. Operandy logických instrukcí jsou dva vstupní registry nebo vstupní registr a 16-bitová konstanta. Narozdíl od aritmetických instrukcí je tato konstanta roztažena nulami na 32 bitů. Dále je možné využít 16-bitový operand, který posune hodnotu na horních 16 bitů operandu, přičemž spodní bity jsou doplněny nulami. Do této skupiny patří instrukce andhi, orhi a xorhi.

Instrukce přesunu

Instrukce přesunu se využívají ke kopírování hodnot ze zdrojového registru nebo konstanty do jiného registru. Jsou implementovány jako pseudoinstrukce, které využívají jiné instrukce. Do této skupiny instrukcí patří:

- mov – přesun hodnot mezi registry, využita instrukce `add`
- movi – přesun znaménkové 16-bitové konstanty do registru, využívá instrukci `addi`
- movui – přesun bezznaménkové 16-bitové konstanty do registru, instrukce `ori`
- movhi – přesun horní poloviny bitů konstanty do registru, využita instrukce `orhi`

Instrukce porovnání

Instrukce porovnání provádí porovnání mezi dvěma zdrojovými registry nebo zdrojovým registrem a 16-bitovým znaménkovým operandem. Na základě vyhodnocení podmínky zapisují jedničku nebo nulu do cílového registru. Další instrukce porovnání jsou implementovány pomocí pseudoinstrukcí využívající opačného porovnání a obrácení pořadí registrů. Do této skupiny patří instrukce:

- cmpeq – znaménkové porovnání rA == rB
- cmpne – znaménkové porovnání rA != rB
- cmpge – znaménkové porovnání rA >= rB, instrukce použita v pseudoinstrukci `cmple`
- cmpgeu – bez znaménkové porovnání rA >= rB, použita pri implementaci `cmpleu`
- cmplt – znaménkové porovnání rA < rB, instrukce použita v pseudoinstrukci `cmpgt`
- cmpltu – bez znaménkové porovnání rA < rB, použita v pseudoinstrukci `cmpgtu`

Instrukce posun a rotace

Instrukce posunu a rotace mají formát instrukcí R-typ. Počet bitů rotace nebo posunu se specifikuje ve vstupním registru nebo 5-bitové konstantě.

Instrukce skoku

Vykonávání programu lze měnit pomocí skokových instrukcí, které dělíme na skoky podmíněné a nepodmíněné. Mezi podmíněné skoky patří například instrukce `beq`, `bne`, **bge** a další. Tyto instrukce vykonávají skok na základě porovnání dvou vstupních registrů. Adresa skoku je dána součtem hodnoty programového čítače a ofsetu.

Nios II dále obsahuje nepodmíněné skoky dané dvěma instrukce pro volání podprogramu `call`, `callr`, které využívají pro návrat z podprogramu instrukci `ret`. Dalšími nepodmíněnými skoky v procesoru jsou `jmp` a `jmpi`. Absolutní adresa skoku je dána hodnotou vstupního registru nebo 26-bitovou konstantou.

4.4 Registry

Procesor Nios II má 32 registrů s bitovou šířkou 32 bitů. Některé z těchto registrů jsou určeny pro konkrétní účely a mají vlastní názvy, které jsou zařazeny do assembleru, jak je uvedeno v tabulce 4.1. Dále obsahuje 32 řídících registrů s šířkou 32 bitů, které se využívají pro účely kontroly. Čtení a zápis do těchto registrů se provádí za pomoci speciálních instrukcí rdctl a wrctl.

Registry	Jméno	Funkce
r0	zero	0x0000
r1	at	Pomocný registr
r2		
r3		
.		
.		
.		
r23		
r24	et	Dočasná výjimka
r25	bt	Dočasná značka
r26	gp	Globální ukazatel
r27	sp	Zásobníkový ukazatel
r28	fp	Základní ukazatel
r29	ea	Dočasná návratová hodnota
r30	ba	Značka návratové hodnoty
r31	ra	Návratová adresa

Tabulka 4.1: Registrové pole

Kapitola 5

Model na úrovni instrukční sady

První krokem v praktické části bakalářské práce bylo vytvoření popisu instrukční sady jádra procesoru NIOS II v jazyce CodAL. Pomocí instrukčního modelu lze automaticky generovat v prostředí Codasip Studia nástroje pro překladač C/C++ popsaný v kapitole 3.1.

Při vytváření instrukčního modelu bylo nutné se vyhnout velkému množství redundantního kódu, proto jsem zvolil rozdělení instrukcí do jednotlivých skupin podle typu operace. Toto je vhodné u mnoha instrukcí, protože se stejné operace opakují a mění se pouze operandy. To nastává převážně u aritmeticko-logických instrukcí např. u ADD, ADDI nebo OR, ORI, ale také u instrukcí nad pamětí (STB, STH, STW), kdy adresa a žádost přístupu k paměti se nemění, pouze offset. To přináší výhody snazší opravy chyb při ladění, přehlednosti navrhovaného modelu a rychlejší kompilace modelu.

5.1 Platforma

Platforma, jak již bylo zmíněno v teoretické části 3.2, obsahuje specifikaci paměťového prostoru, rozhraní, zapojení komponent a periferie.

Paměť

Nios II má jeden adresový prostor jak pro data, tak i pro instrukce programu. Tento adresový prostor definujeme pomocí argumentů jako jsou délka slova, zarovnaný či nezarovnaný přístup a velikost paměti, která je omezena na 512 kB, i když Nios II/s může obsahovat adresový prostor až 2 GB. Toto omezení je z důvodu délky doby generování nástrojů.

Paměť komunikuje s jádrem procesoru pomocí rozhraní (*interface*) a jednotlivá rozhraní paměti na platformě jsou typu SLAVE. Typicky jsou to rozhraní `if_fetch` pro načítání jednotlivých instrukcí programu a `if_ldst` pro datové operace nad pamětí pomocí instrukcí LOAD a STORE.

Propojení rozhraní probíhá pomocí příkazu `connect`. Implementace paměti na platformě a propojení s modelem procesoru Nios II je znázorněno na zdrojovém kódu 5.1.

```
memory mem {                            // Parametry paměti, definující
    size = MEM_SIZE;                    // vlastnosti paměti např.
    bits = {ADDR_W, WORD_W, LAU_W}; // velikost  paměti, endianita,
    endianness = ENDIAN;           // zarovnání atd.
    latencies = {1, 1};
    unaligned = true;

    interface if_fe           // Rozhraní pro načítání instrukcí
    {                         // z paměti typu SLAVE, podporuje
        type = MEMORY:SLAVE; // pouze čtení
        flag = R;
    };

    interface if_ldst         // Rozhraní pro práci s daty typu SLAVE
        .
    {                         // Rozhraní umožňuje číst a zapisovat
        type = MEMORY:SLAVE; // z paměti.
        flag = RW;
    };
};

connect nios2.if_fe => mem.if_fe;       // Propojuje jádro procesoru
connect nios2.if_ldst  => mem.if_ldst; // NIOS~II s pamětí
```

Zdrojový kód 5.1: Paměťový prostor a jeho propojení s modelem procesoru

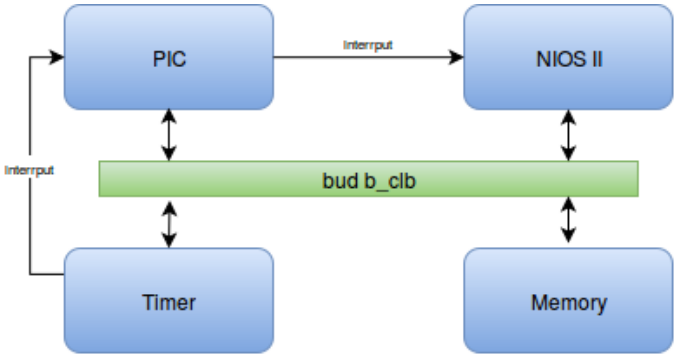
Komponenty

Na platformě jsou dále definovány připojené komponenty (neboli pluginy). Komponenty mají hlavní rozhraní napsané v jazyce C nebo C++, tím je vytvořen most mezi simulátorem a implementací komponentů využívaných u instrukčních modelu. U modelu na úrovni cyklů je komponenta popsaná ve VHDL nebo Verilog jazyce. Model procesoru Nios II využívá pro přerušení dvě komponenty Timer a PIC (*Programmable Interrupt Controller*), které jsou připojeny k jádru procesoru, společně s pamětí přes sběrnici:

- **Timer** – je jednoduchý časovač, který posílá žádost o přerušení po vykonání stanoveného počtu hodinových cyklů. Přerušení se povolují buď jednorázově, nebo v pravidelných intervalech.
- **PIC** – je periferie, ke které může být připojeno více zdrojů přerušení. To poskytuje jediný výstupní signál, který jádro procesoru informuje, jaké přerušení má být na řadě.

Sběrnice

Pro přístup k periferiím jsem zvolil sběrnici s vlastním dekodérem adres. To umožňuje jednodušší adresování a v případě potřeby lze doplnit model o další periferie. Propojení jednotlivých periferií je znázorněno na obrázku 5.1.



Obrázek 5.1: Zapojení periferií na platformě přes sběrnici

5.2 Registry a porty

Jeden z nejdůležitějších registrů procesoru je programový čítač (PC), který má bitovou šířku 32 bitů a obsahuje adresu aktuálně prováděné instrukce. Během vykonávání instrukce se obsah programového čítače mění na adresu následující instrukce, která má být vykonána. Obsah PC se také mění během vykonávání skoku nebo instrukcemi větvení.

Nios II obsahuje dále 32 obecných registrů, které jsou součástí registrového pole `rf_gpr`, specifikované klíčovým slovem `arch`. Hodnota takového registru může být externě viditelná pro sémantiku, v jiném případě registry bez tohoto označení uchovávají dočasné hodnoty během provádění sémantiky. Dále některé registry z registrového pole jsou pojmenovány a využívají se při vykonávání určitých operací (popsány v tabulce 4.1). To umožňuje instrukci volání (*call*) se navracet na původní adresu přes registr `RA`.

Řídící registr `r_state` je 1-bitový registr. Hodnota po resetu je nula, to znamená přerušení není povoleno. Tento registr se nastavuje pomocí speciálních instrukcí INT_EN a INT_DIS.

Port `p_irq` je 1-bitový port, který je externím výstupním rozhraním procesorového jádra a využívá se pro přenos informace o přerušení.

5.3 Modelování instrukcí

Instrukční sada procesoru Nios II obsahuje tři základní formáty instrukcí I-typ, R-typ a J-typ, popsány v kapitole 4.2. Pole OP v instrukci procesoru určuje hlavní třídu operačních kódů. Většina hodnot OP je určena pro instrukce typu I. Výjimkou je operační kód `OP=0x00` pro instrukci volání typu J a další `OP=0x3A` je použit pro všechny instrukce typu R, kde jsou jednotlivé instrukce rozlišeny pomocí rozšířeného operačního kódu (OPX). Tyto operační kódy jsou definovány ve složce *isa_opcodes.hcdl*.

Dále si uvedeme postup pří vytváření instrukce BEQ (*branch if equal*), kterou jsem použil při modelování všech instrukcí procesoru. Instrukce BEQ je dlouhá 32 bitů a řadí se mezi instrukce skoku. Skok se provádí na základě porovnání hodnot dvou operandů. Pokud jsou si rovny, provede se skok na adresu danou součtem aktuálního programového čítače a 16-bitovým offsetem. Formát instrukce je vidět na obrázku 5.2.

Nejprve bylo nutné vyhledat všechny instrukce, které používají stejné operandy jako instrukce BEQ. Dále jsem si vytvořil množinu, která sdružuje použité operační kódy a vložil

beq

Instruction	branch if equal
Operation	if (rA == rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`beq rA, rB, label`

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x26					

Obrázek 5.2: Formát instrukce BEQ

do ní elementy popisující operační kódy instrukcí. Tyto elementy jsou definovány pomocí makra, do kterého se vkládá název elementu, název instrukce v assembleru a operační kód. Pro popis operačních kódů instrukce BEQ jsem vytvořil následující elementy a množiny. Na příkladu je vidět pouze část množiny instrukcí, které byly využity, zbylé se definují podobným způsobem.

```
DEF_OPC(beq, "beq", OPC_BEQ)     // Skok při rovnosti
DEF_OPC(bne, "bne", OPC_BNE)     // Skok, pokud se nerovná

// vytvořený element pomocí makra
element opc_beq{
  assembler{"beq"};
  binary{OPC_BEQ:bit[OPC_W]};
  result{OPC_BEQ;};
};

element opc_bne{
  assembler{"bne"};
  binary{OPC_BNE:bit[OPC_W]};
  result{OPC_BNE;};
};

set opc_branch = opc_bep, opc_bne; // Vytvoření kolekce elementů
```

Zdrojový kód 5.2: Definice makra a vytvořený element instrukce BEQ

K jednotlivým instrukcím jsem vytvořil elementy pro operační kód a ty spojil do jedné množiny s názvem `opc_branch`. Dále jsem vytvořil element sjednocující podmíněné skoky.

```
element i_branch
{
    use opc_branch as opc;
    use gpr_all as  srcA, srcB;
    use rel_addr16;

    assembler { opc srcA "," srcB "," rel_addr16 };
    binary { srcA srcB rel_addr16 opc};

    semantics
    {
        if( cond_branch(opc,rf_gpr_read(srcA),rf_gpr_read(srcB)) )
        {
            r_pc = r_pc + rel_addr16;
        }
    };
};

set isa +=  i_branch;
```

Zdrojový kód 5.3: Element podmíněné skoky

Každá instrukce procesoru Nios II používá pouze jeden typ operandu, proto není nutné vytvářet další kolekce elementů. Elementy operandu se vytváří podobným způsobem jako elementy operačního kódu.

V **semantics** sekci elementu `i_branch` používám fukci *cond_branch*, která vrací výsledek porovnání dvou operandů.

Vytvořil jsem také dvě funkce *rf_gpr_read* a *rf_gpr_write* pro čtení a zapis do registrového pole. Jelikož BEQ je instrukce podmíněného skoku, funkci pro zápis do registrového pole nevyužívám. Zápis probíhá pouze do registru programového čítače. Na základě vyhodnocení podmínky funkce *cond_branch* se do registru zapisuje součet programového čítače a offsetu.

Podobným způsobem jsou implementovány zbylé instrukce instrukčního modelu. Všechny namodelované instrukce jsou sdruženy do jedné množiny `isa`, která je použita při generování nástrojů a simulací.

Pseudoinstrukce

Další možností, jak modelovat instrukce, je pomocí vytvoření pseudoinstrukcí (aliasy), které využívají operační kód a sémantiku jiné instrukce. Pseudoinstrukce se liší od původních instrukcí názvem v assembleru a například záměnou operandu bitové části instrukce. To umožňuje využívat již implementované instrukce.

Procesor Nios II podporuje vytváření mnoha aliasů, například instrukce pro přesun MOV, která využívá již implementovaný aritmeticko-logický element s instrukcí ADD. Alias tvoří pouze zdrojový a cílový registr, kdy na místo druhého zdrojového registru je přiřazen nulový registr. V podstatě se provede součet konstanty s nulou a výsledek se uloží do cílového registru, tím je zajištěn přesun. V assembleru je instrukce zapsána jako MOV. Ukázka implementace aliasu znázorněna na zdrojovém kódu 5.4.

```
element i_mov_alias : assembler_alias(i_arithmetic)
{
    use gpr_all as  srcA, dstC;
    assembler { "mov" dstC "," srcA };
    binary { srcA GPR_ZERO:bit[5] dstC OPX_ADD:bit[6] 0:bit[5]
       OPC_R_TYPE:bit[6] };
};
```

Zdrojový kód 5.4: Element podmíněné skoky

Rozdíly od původního procesoru

V modulu jsem implementoval několik instrukcí, které jsou nutné v procesoru, ale nejsou specifikovány v dokumentaci. Jednou z instrukcí s rezervovaným operačním kódem 0x02 je HALT, který zajišťuje ukončení simulace programu.

Podpora programového přerušení je v modelu implementována trochu odlišně. Komponenty využité pro přerušení jsou popsány na začátku kapitoly. Požadavek na přerušení je generován časovačem a dále zpracován komponentou *PIC*, která nastavuje port `p_irq`. Pokud je v procesoru nastaven řídící registr `r_stav` provede poté se přerušení. Pro podporu přerušení bylo nutné implementovat tyto instrukce:

- `call_int` – instrukce je v podstatě stejná, jak instrukce `call`, s tím rozdílem, že provádí skok vždy na adresu 0x80 a ukládá návratovou adresu do registru R24. Tento registr má vyhrazenou funkci v „rychlé“ architektuře, a ta se v modelovaném jádře nevyužívá, proto jsem zvolil právě registr R24.
- `ret_int` – instrukce pro návrat z rutiny přerušení.
- `en_int` – instrukce pro nastavení řídícího registru.
- `dis_int` – intrukce pro nulování řídícího registru.

Dále jsem implementoval systémovou instrukci `syscall`, která umožňuje přístup k systémovým funkcím. V jazyce CodAL se používá funkce *codasip_syscall*, která předává do vyhrazeného registru R2 adresu struktury požadované systémové funkce.

Kapitola 6

Hardwarový model

Po dokončení instrukčního modelu je dalším krokem praktické části bakalářské práce návrh hardwarového modelu. Protože není k dispozici návrh mikroarchitektury jako celku v dokumentaci Nios II od Altery, ale pouze popis částí, ze kterých se skládá, bylo nutné vytvořit vlastní návrh architektury procesoru. Nejdůležitější požadavky, které byly kladeny na procesor Nios II z pohledu implementace ve vývojovém prostředí Codasip Studia, jsou rychlost zpracování instrukcí a velikost místa na čipu. Jen tak je možné vytvořit konkurenční řešení, které může být lehce přizpůsobené různým potřebám v IoT, narozdíl od fixních 3 konfigurací v podání architektury.

Nejprve při návrhu CA modelu bylo nutné rozhodnout, kterou architekturu budu modelovat. Jelikož Nios II má jednu instrukční sadu, která podporuje tři mikroarchitektury „rychlou“, „ekonomickou“ a „standardní“, které se liší v rychlosti zpracování instrukcí, velikostí jádra na čipu a některými řídícími instrukcemi, rozhodl jsem se modelovat architekturu typu „standard“. Architektura typu „rychlá“ má 6-stupňovou zřetězenou linku, jsou zde zapojeny komponenty MMU (*Memory management unit*) a MPU (*Memory protection unit*), tato architektura má větší výpočetní výkon, ale na FPGA jádro procesoru zabírá více logických jednotek. Zatímco architektura typu „ekonomická“ s 1-stupňovou zřetězenou linkou vykonává každou instrukci po dobu 6hodinových cyklů, čímž je výrazně zpomalena výpočetní schopnost procesoru. Výhodou je velikost jádra, které využívá méně jak 700 logických elementů na FPGA (detailnější popis architektur se nachází v kapitole 4).

Implementace procesoru se bude věnovat architektuře „standard“. Je to kompromis mezi rychlostí a velikostí jádra na čipu.

Linka je rozdělena do následujících pěti stupňů:

- Načtení instrukce (FE)
- Dekódování instrukce (ID)
- Výpočet instrukce (EX)
- Paměť (MEM)
- Uložení (WB)

První stupeň zajišťuje zasílání požadavků na instrukce z programové části paměti a také aktualizaci programového čítače.

Ve druhém stupni se dokončují požadavky na získání instrukce z předchozího stupně. Jednotlivé instrukce se dekódují pomocí instrukčního dekodéru a vkládají do další fáze.

Ve třetím stupni je umístěna aritmeticko-logická jednotka, komparátor a násobička, které mají vstupy a výstupy 32-bitové šířky. Při vykonávání instrukce se provedou výpočty na všech blocích, jelikož jsou operandy přivedeny ke každému bloku. Příslušný výsledek instrukce je vybrán na výstupu stupně.

Čtvrtý stupeň zajišťuje přístup do datové části paměti. To znamená zasílání požadavku pro čtení nebo zápis do paměti. Tento stupeň se dále využívá pro nastavení adresy skokových instrukcí a u podmíněných skoků je podmínka vyhodnocena v předchozím stupni linky.

Páty stupeň slouží k dokončení čtení a zápisu dat do paměti, dále k ukládání výsledků instrukcí do registru.

Ve zřetězené lince můžeme rozlišit datové signály, které se využívají pro přenos dat mezi bloky a jednotlivými stupni linky. Tyto signály mají 32-bitovou šířku, proto 16-bitové operandy jsou znaménkově nebo bezznaménkově roztaženy. Dále rozlišujeme řídící signály, které slouží k ovládání funkčních bloků. Jedná se například o komparátor, ALU, paměť nebo multiplexory. Při dekódování instrukce jsou tyto signály nastaveny tak, aby zajišťovaly kontrolu nad jednotlivými bloky v aktuálním stupni. V každém hodinovém cyklu se signály posouvají o jeden stupeň vpřed.

6.1 Implementace

Implementace CA modelu v jazyce CodAL umožňuje snadně rozdělit zřetězenou linku do jednotlivých stupňů, proto základní strukturu modelu podle návrhu nebylo obtížné popsat. Ve složce *ca_resources.acdl* jsou definovány jednotlivé stupně linky a ke každému stupni sada registrů a signálů, které příslušný stupeň ovládají (ukázka na zdrojovém kódu 6.1). Poté se definují pro každý stupeň linky události, které popisují jejich chování. V každém stupni se může vyskytnout více událostí, ale musí se dát pozor, aby se provedly ve správném pořadí.

```
// Registry třetího stupně linky
register   bit[ALU_OP_W] r_ex_alu_op { pipeline = pipe.EX; };
register   bit[3] r_ex_cond_op   { pipeline = pipe.EX; };
register bit[3] r_ex_mem_op { pipeline = pipe.EX; };

// Definice jednotlivých stupňů linky
pipeline pipe
{
    FE, ID, EX, MEM, WB
};
```

Zdrojový kód 6.1: Definice zřetězené linky a registrů

Hlavní události

Důležité události procesoru v jazyce CodAL zajišťují základní kontrolu jak u instrukčního, tak u hardwarového modelu. Mezi tyto události patří:

- main – událost je vyvolaná v každém hodinovém taktu, který určuje, v jakém pořadí se vykonávají jednotlivé stupně linky. Zpravidla je to od posledního stupně linky k prvnímu, aby se zajistilo případné pozastavení linky nebo jiných výjimek.

- reset – je proveden na začátku každého programu, kdy inicializuje všechny registry registrového pole na nulu.
- halt – je vykonán při ukončení programu a ukládá návratovou hodnotu provedeného programu do registru R3.

Dekodér

Dekódování instrukcí probíhá v druhém stupni linky pomocí události *ID*, která dále zajišťuje detekci požadavku na "vyčištění" a linky přerušení. K čištění linky dochází v případě, že instrukce v prvním stupni byla pozastavena, a do dekodéru je vložena instrukce NOP. Podobný případ nastává při detekci přerušení, kdy je do dekodéru vložena instrukce CALL_INTERRUPT s příslušnou adresou skoku. Tato instrukce ukládá návratovou adresu naposledy načtené instrukce.

Procesor Nios II obsahuje dva 6-bitové operační kódy instrukcí, které se spojí v jeden 12-bitový signál přivedený do dekodéru. Dekodér na základě spodní šestice bitů rozhoduje, o jaký formát instrukce se jedná a poté provede dekódování instrukce podle operačního kódu.

Ve druhém stupni linky je dále řešené ošetření datových hazardů, které jsou popsané v podkapitole 6.2.

Popis stupně linky

Jak již bylo zmíněno na začátku kapitoly, každý stupeň linky se skládá z několika událostí. Ty můžeme rozdělit na vstupní, které získávají signály z předchozí linky a výstupní události, které již zpracované signály předávají dál (příklad implementace ID stupně 6.2). Funkcionalita jednotlivých událostí je popsána v sekci *semantics*. Každá událost může aktivovat jinou událost nebo dekodér, to umožňuje vytvářet větší celky. Použití jiných událostí je nutné provést ve správném pořadí. Při používaní jiných událostí je nutné zajistit pořadí vykonání jednotlivých instrukcí, jinak by simulátor nevěděl, kterou událost má dále vykonávat. Toto zajišťuje sekce *timing*.

```
event id : pipeline(pipe.ID)
{
    use dec;            // Dekodér
    use id_output;      // Výstup dekódovacího stupně
    semantics
    {
    ...         // Rozdělení instrukce na jednotlivé části
    ...         // jako je operační kód, operandy a registry
    };
    decoders
    {
        { dec(s_id_opcode); }
    };
    timing
    {
        id_output();
    };
};
```

Zdrojový kód 6.2: Druhý stupeň zřetězené linky

6.2 Ošetření hazardu

Při vytváření popisu modelu procesoru jsem musel řešit datové hazardy způsobené pořadím, ve kterém jsou instrukce využívány. Jelikož procesor využívá jeden adresový prostor zapojený pomocí dvou rozhraní k jádru, nebylo tedy nutné se zabývat strukturálními hazardy.

Linka používá pro šíření informace o vzniklém hazardu signály typu *clear* a *stall* (např. *fe_stall* – pozastavení prvního stupně, *id_clear* – čištění dekódovacího stupně, ...). Pokud vznikne v nějakém stupni hazard a jeho řešením je pozastavení linky, pak se využije signál *stall*. Pokud zároveň tento hazard ovlivňuje poslední stupeň linky, aktivuje se signál *clear*, který uvolní registry tohoto stupně.

Díky uspořádání stupňů v lince pro čtení a zápis rozlišujeme pouze jeden typ datového hazardu a to RAW, který ovlivňuje všechny registry a adresový prostor při čtení. Tento typ hazardu lze řešit pomocí pozastavení linky do doby, než je výsledek instrukce zapsán do registru. Řešení hazardu tímto způsobem využívá signál *clear*, jinak by instrukce aktivující pozastavení byla provedená několikrát. Dalším způsobem řešení hazardu je pomocí techniky zvané forwarding, kdy jsou přidány datové signály z pozdějších stupňů linky do třetího stupně (EX). To umožňuje například aritmeticko-logické jednotce pracovat s aktuálními daty ještě před dokončením předchozí instrukce a jejich zápisem do registru či paměti. Tuto techniku řešení hazardu jsem se snažil využívat, protože maximálně využívá potenciál zřetězené linky ve srovnání s pozastavením, které snižuje výkon.

Další datový hazard nastává v případě využití instrukce LOAD a následně instrukce pracující s těmito daty. Jelikož načtení dat z paměti je dokončeno až v posledním stupni, je nutné pozastavit linku, než budou data zapsány do registru. Další řešení je pomocí nastavení latence modelu. Překladač tuto hodnotu zohledňuje při generování assembleru programu, to znamená, že při vzniklém hazardu způsobeného instrukcí LOAD jsou do linky vloženy prázdné instrukce (NOP).

Kapitola 7

Testování a funkční verifikace

7.1 Testování

Jelikož Codasip Studio umožňuje generovat nástroj překladač jazyka assembler již při popsání základních instrukcí jako jsou NOP a HALT, mohl jsem testovat instrukční sadu během návrhu. Na jednoduchých programech psaných v assembleru jsem ověřil funkčnost jednotlivých instrukcí. Tyto jednoduché programy jsem simuloval a odkrokoval pomocí debug režimu a následně provedl případnou opravu chyb v návrhu procesoru.

Po dokončení a odladění základní implementace instrukční sady jsem vygeneroval překladač jazyka C. Ten umožňuje překlad zdrojového kódu jazyka C na assembler (jednotlivé nástroje jsou popsány v kapitole 3.1). Před spuštěním programu jazyka C je nutné nastavit soubor crt0.s, který je napsaný v assembleru a obsahuje startovací sekvenci. Startovací sekvence programu obsahuje inicializaci zásobníku, volání těla funkce, návratové hodnoty a ukončení programu.

K poslední fázi testování mi byla firmou Codasip zpřístupněna základní testovací sada, která obsahovala celkem 703 testů. Během testování v prostředí Codasip Studia se kontrolovalo, zda byly dodrženy všechny požadavky, například problém při kompilaci testu, problém při linkování programu nebo vypršení času testu.

Při testování instrukčního modelu se vykonaly správně všechny testy, zatímco u modelu na úrovni cyklů bylo nutné zvýšit čas vykonávaní testu až na 120 sekund, protože programy, které byly překládány bez optimalizace (-O0), vykonávaly výpočet více taktů a nedokončily se. Po zvýšení času vykonávání testu se provedly správně všechny testy z testovací sady.

7.2 Verifikace

Funkční verifikace je metoda, která ověřující zda generovaný popis procesoru v HDL jazyce (ve verifikací zvaný DUT – Design Under Test) funkčně odpovídá popisu procesoru na úrovni jazyka CodAL konkrétně IA popisu (ve verifikaci jako referenční nebo golden model).[4]

Pokrytí (*coverage*) se používá k vyhodnocení průběhu funkční verifikace. Pokrytím se zjišťuje, zda byl zdrojový kód (model procesoru, komponenta, ...) úspěšně vykonán a u funkční pokrytí, zda byly použity všechny požadované funkce a instrukce procesoru.

Celé verifikační prostředí je generováno vývojovým prostředím Codasip Studia. Konkrétně z CA modelu se generuje syntetizovatelné RLT (tj. DUT) a instrukční model je použit ke generování referenčního modelu v C++.

Funkční verifikaci jsem provedl na stejných programech z testovací sady. K jednotlivým zdrojovým kódům programů se po testování v Codasip Studiu vygeneroval spustitelný soubor (ve formátu .xexe), který jsem následně vložil do verifikačního prostředí. Verifikační prostředí je navrhnuto tak, že se programy nahrají do RTL a referenčního modelu.

Výsledky automatické funkční verifikace jsem vložil na přiložené CD. Jak jde vidět na ukázce ze souboru 7.2, funkční verifikace proběhla úspěšně, kdy „GM cycles“ vyjadřuje počet cyklů referenčního modelu, „DUT cycles“ vyjadřuje počet cyklů DUT a poslední sloupeček výsledek verifikace daného programu.

#executable;	GM cycles;	DUT cycles;	verification result
;nios2_platform;xexes/900409-1.c.xexe;	153;	234;	ok
;nios2_platform;xexes/920202-1.c.xexe;	48;	72;	ok
;nios2_platform;xexes/920409-1.c.xexe;	42;	64;	ok
;nios2_platform;xexes/920410-1.c.xexe;	25;	34;	ok
;nios2_platform;xexes/920411-1.c.xexe;	239;	347;	ok
;nios2_platform;xexes/920428-1.c.xexe;	65;	89;	ok
;nios2_platform;xexes/920429-1.c.xexe;	83;	109;	ok
...			

Tabulka 7.1: Výsledky funkční verifikace ze souboru

Jelikož funkční verifikace proběhla úspěšně, dalším krokem bylo zjistit odlišnosti referenčního modelu a DUT pomocí pokrytí. Celková hodnota pokrytí je 78.8%. V tabulce 7.2 vidíme, že některé částí CA modelu nejsou zcela pokryty. To je způsobeno například u větví chybějící implementace aliasů podmíněných skoků. Dále nízkou hodnotu pokrytí konečných automatu lze zvýšit pomocí Pseudo-Random generátoru ASIP testů, který pokryje kombinace instrukcí, které nebyly použity v testovací sadě. Ten umožňuje náhodně generovat testy, který je následně možné využít k otestování ASIPu. Tímto způsobem lze odhalit, jaké instrukce, popřípadě kombinace instrukcí, nejsou pokryty.

Enabled Coverage	Active	Hits	Misses	Weight	Covered [%]
Stmts	2100	2017	83	1	96.0
Branches	2078	1789	289	1	86.0
UDP Condition Rows	0	0	0	1	100.0
UDP Expression Rows	0	0	0	1	100.0
FSMs				1	83.3
States	4	4	0	1	100.0
Transitions	9	6	3	1	66.6
Toggle Bins	0	0	0	1	100.0

Tabulka 7.2: Tabulka pokrytí CA modelu. Stmts = výroky, Branches = větve (if, else, case), UDP Condition Rows = podmínky, UDP Expression Rows, FSMs = stavy a přechody konečných automatů, Toggle Bins = přechody mezi 0 a 1 na jednotlivých bitových pozicích signálu

7.3 Syntéza HDL

Poslední fázi testování hardwarového modelu procesoru Nios II jsem provedl na desce Nexys4 DDR. Cílem bylo spustit aplikaci, která zobrazovala na sedmisegmentovém číselníku aktuální čas. Po vložení aplikace do modelu jsem provedl generování HDL popisu Codasip Studiem a následnou syntézu pomocí nástroje Xilinx ISE. Následně vytvořený bitstream[1] jsem nahrál na vývojovou desku Nexys 4 DDR s FPGA typu Artix XC7A100T-CSG324.

Dále lze ze syntézy zjistil základní informace o hardwarovém modelu, které nám ukazují využití zdrojů FPGA.

- Frekvence – 102.63 MHz
- Počet LUT[2] – 2407
- Počet registrů – 760

Frekvence jádra je pouze orientační údaj, protože nezohledňuje zpoždění přístupu do paměti a dále závisí na nastavené optimalizaci, proto reálný model procesoru Nios II/s dosahuje přibližně maximální rychlosti 165 MHz. Plocha procesoru je také podstatně větší než reálný model, který by měl odpovídat méně jak 1400 logických elementů, výsledky mohou záviset také na typu FPGA. Tato omezení lze řešit optimalizacemi modelu.

Výhoda modelování procesoru pomocí jazyka CodAL v Codasip Studiu je při výskytu chyb nebo úpravě modelu. Model procesoru je tak možné upravit na vyšší úrovni abstrakce a následně možné vygenerovat příslušný HDL popis procesoru. V jiném případě by bylo nutné tyto změny provádět na nejnižší úrovni abstrakce tzn. úpravu popisu procesoru ve VHDL jazyce. To umožňuje vývojáři snadněji a rychleji provádět změny v návrhu procesoru a výrazně zkrátit dobu vývoje.

[1] Bitstream slouží k popisu konfiguračních dat, které jsou nahrány na FPGA
[2] LUT (*LookUp Table*)

Kapitola 8

Závěr

Cílem této bakalářské práce bylo navrhnout model procesoru Nios II v jazyce pro popis architektur CodAL. Návrh procesoru se skládal z vytvoření dvou dílčích částí, jednak instrukčního modelu, který popisuje jednotlivé instrukce instrukční sady procesoru a hardwarového modelu. Tvorba hardwarového modelu zahrnovala celkový návrh architektury procesoru s využitím zřetězené linky, implementaci modelu, odladění testovací sadou a ověření pomocí funkční verifikace.

Během práce jsem musel průběžně upravovat svůj model v závislosti na zveřejňovaní novějších verzí vývojového prostředí Codasip Studia. To zahrnovalo například úpravu hardwarového modelu, protože se pozměnilo časování zápisu signálů do registru.

Testování instrukčního i hardwarového modelu proběhlo úspěšně na testovací sadě celkem 703 programů na třech stupních optimalizace. Další fáze testování byla verifikace celkového návrhu hardwarového modelu. Verifikace probíhala automaticky srovnáním instrukčního a hardwarového modelu a nebyly odhaleny žádné chyby. Po úspěšné verifikaci jsem zjišťoval pokrytí instrukční sady v hardwarovém modelu, které celkově činí 78.8 %. Je to způsobeno chybějícími instrukcemi v návrhu modelu. Proto v další fázi vývoje bych se zaměřil na doplnění návrhu hardwarového modelu o aliase instrukcí větvení a porovnání, které nejsou pokryty instrukčním modelem. Další možností vývoje by bylo zaměření se na optimalizaci hardwarového modelu pomocí implementace statické predikce skoku, která by zajistila snížení latence skoků. Z pohledu ladění přímo v hardwaru (FPGA) je možné model rozšířit o JTAG ladící režim.

Hardwarový model po úspěšné verifikaci byl ještě otestován na vývojové desce Nexys 4 DDR s FPGA Artix pomocí demonstrační aplikace.

Literatura

[1] Altera: Nios II Gen2 Processor Reference Guide. https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpunii5v1gen2.pdf, 2015.

[2] Codasip Ltd.: Codal Language Reference Manual. 2016.

[3] Codasip Ltd.: Codasip Studio Technical Reference Manual. 2016.

[4] Codasip Ltd.: Codasip Studio User Guide. 2016.

[5] John L. Hennessy, David A. Patterson: *COMPUTER ARCHITECTURE A Quantitative Approach*. San Francisco : Morgan Kaufmann, 2012, iSBN 0-12-370490-1.

[6] Masařík, K.: *Systém pro souběžný návrh technického a programového vybavení počítačů : disertační práce*. Dizertační práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2008, iSBN 978-80-214-3863-7.

[7] Schwarz, J.; Růžička, R.; Strnadel, J.: Mikroprocesorové a vestavěné systémy. Studijní opora k předmětu Mikroprocesorové a vestavěné systémy., 2006.

Příloha A

Obsah CD

Kořenový adresář disku obsahuje složky:

- `/doc` – složka s technickou zprávou
- `/model` – model procesoru NIOS II
  - `/nios2` – složka s IA a CA modely
  - `/nios2_platform` – platforma procesoru
- `/test` – výsledky testů a verifikací
  - `/texts` – složky s výsledky testů testovací sady
  - `/synthesize` – výstupní zpráva syntézy
  - `/fve` – výsledky funkční verifikace
- `/tex` – zdrojové text bakalářské práce