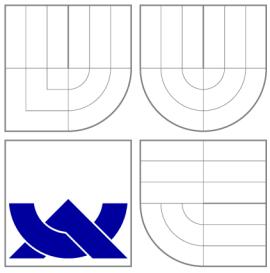
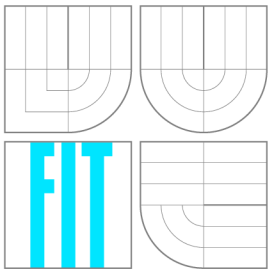


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VERIFIKACE ASIP ZALOŽENA NA FORMÁLNÍCH TVRZENÍCH

ASSERTION-BASED VERIFICATION OF ASIP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB ŠULEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARCELA ŠIMKOVÁ

BRNO 2015

Abstrakt

Tato práce představuje koncept pro ověřování správnosti procesorů s aplikačně-specifickou instrukční sadou (ASIP) pomocí verifikace založené na formálních tvrzeních. Koncept je implementován v jazyku SystemVerilog Assertions jako součást verifikačního prostředí vytvořeného v nástroji Codasip Framework. Implementovaný koncept je simulován nástrojem QuestaSim na procesoru Codix RISC. Hlavním výsledkem práce je koncept ověřování, který může být součástí systému automatizujícího návrh procesorů, a který je použitelný pro různé typy procesorů.

Abstract

This thesis introduces the concept of assertion-based verification of application-specific instruction set processors (ASIPs). The proposed design is implemented in SystemVerilog Assertions language as a part of verification environment created using Codasip Framework. The implemented concept is simulated in QuestaSim tool using model of Codix RISC processor. Main outcome of this thesis is the verification concept usable not only on other processors, but as a part of system that automates the processor design as well.

Klíčová slova

SystemVerilog Assertions, verifikace založena na formálních tvrzeních, procesor s aplikačně-specifickou instrukční sadou, verifikační prostředí

Keywords

SystemVerilog Assertions, assertion-based verification, application-specific instruction set processor, verification environment

Citace

Jakub Šulek: Verifikace ASIP založena na formálních tvrzeních, diplomová práce, Brno, FIT VUT v Brně, 2015

Verifikace ASIP založena na formálních tvrzeních

Prohlášení

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

.....
Jakub Šulek
24. května 2015

Poděkování

Chcel by som sa poďakovať vedúcej mojej práce Ing. Marcele Šimkovej a konzultantovi Ing. Zdeňkovi Přikrylovi, PhD. za užitočné rady, za čas a snahu venovanú konzultáciám a za odborné vedenie mojej diplomovej práce. Taktiež by som sa rád poďakoval tímu Codasip za podporu poskytovanú počas tvorby práce.

Obsah

Kapitola 1

Úvod

Verifikácia predstavuje nezávislú činnosť, ktorá vyhodnocuje, či produkt, služba alebo systém vyhovuje špecifikáciám, podmienkam a požiadavkám. V súčasnosti zastáva významné miesto v procese návrhu moderných číslicových systémov, či už ide o formálnu alebo funkčnú verifikáciu. V poslednom desaťročí sme boli svedkami významného záujmu o verifikáciu založenú na fomálnych tvrdeniach, ktorá sa posunula za hranice akademickej diskusie do sféry priemyselnej aplikácie. Dôkazom je štandardizovanie jazykov ako SystemVerilog Assertions či Property Specification Language skupiny Accellera ako aj uplatnenie firiem, ktoré využívajú tento typ verifikácie, napr. Codasip, Synopsis, Mentor Graphics, Cadence či Altera.

S neustále rastúcou komplexnosťou a rozmanitosťou špecializovaných procesorov rastú aj náklady a požiadavky na ich overovanie a testovanie. Tradičné metódy využívajúce nizkoúrovňové popisy sú často príliš nákladné a časovo náročné, pričom využitím moderných prístupov a ich začlenením do celého procesu návrhu je možné dosiahnuť skrátenie návrhového cyklu a úsporu zdrojov. Práve verifikácia založená na formálnych tvrdeniach je jedným z možných prístupov k overovaniu funkčnej správnosti návrhu a predstavuje jednu z najväčších výziev pre súčasné procesory s aplikačne špecifickou inštrukčnou sadou (anglicky Application-Specific Instruction Set Processor, skrátene ASIP) a systémy na čipe (anglicky System on Chip, skrátene SoC).

Hlavným cieľom predkladanej práce je implementácia návrhu, ktorý pomocou verifikácie založenej na formálnych tvrdeniach overuje komponenty procesorov s aplikačne-špecifickou inštrukčnou sadou. Práca predstavuje použité prostriedky, techniky a postupy verifikácie hardvéru v jazyku SystemVerilog a verifikáciu založenú na formálnych tvrdeniach za účelom návrhu koceptu pre overovanie správnosti ASIP-ov.

V práci je na implementáciu a testovanie využitý model procesoru Codix RISC spolu s testovacími aplikáciami od firmy Codasip [16] a verifikačné prostredie vygenerované nástrojmi dodanými taktiež firmou Codasip. Implementácia a jej následné testovanie na modeli procesoru je realizované pomocou simulácie v nástroji QuestaSim spoločnosti Mentor Graphics. Práca využíva návrh vytvorený a prezentovaný v rámci semestrálneho projektu.

Úvod práce popisuje framework firmy Codasip a procesor Codix RISC. Následne sa práca zaoberá verifikáciou hardvérových obvodov a podrobnejšie popisuje verifikáciu založenú na formálnych tvrdeniach a jazyk SystemVerilog Assertions. V ďalšej kapitole sa práca zameriava na verifikáciu pomocou formálnych tvrdení pre procesory ASIP a ich komponenty. Ďalšiu časť práce tvorí analýza súvisiacich prác nasledovaná kapitolou prezentujúcou navrhnuté riešenie overovania správnosti vybraného procesoru. Prácu uzatvárajú kapitoly popisujúce implementáciu navrhnutého riešenia a zhodnotenie vytýčených cieľov.

Kapitola 2

Procesory s aplikačne-špecifickou inštrukčnou sadou

Procesor s aplikačne špecifickou inštrukčnou sadou (anglicky Application-Specific Instruction Set Processor, skrátene ASIP) je komponent používaný v systémoch na čipe (anglicky System-on-a-Chip, skrátene SoC) integrovaných obvodoch. Jeho inštrukčná sada je navrhnutá a prispôsobená pre vykonávanie určitej aplikácie, či triedy aplikácií, ktoré sú spúšťané na procesore. Táto forma špecializácie procesoru prináša prostredníka medzi flexibilitou univerzálnych procesorov a výkonom aplikačne špecifických integrovaných obvodov (anglicky Application-Specific Integrated Circuit, skrátene ASIC). [11]

ASIP-y majú inštrukčnú sadu optimalizovanú na efektívnejšie vykonávanie konkrétnej aplikácie, či triedy aplikácií, čo na jednej strane dovoľuje vytvárať rôzne aplikácie, a na druhej poskytuje výkon a vlastnosti vysoko špecializovaného procesoru. Zvýšená efektivita, na rozdiel od univerzálnych procesorov, však nemusí súvisieť iba s maximalizáciou priemerného výkonu, či zrýchlením vykonávaného kódu. Často požadovanými cieľmi sú aj zmenšenie spotreby energie, znížene nákladov na výrobu, zjednodušenie výrobného procesu, prípadne iné špecifické požiadavky. Pri dosahovaní maximálneho zefektívnenia je podstatné zameranie sa na zdrojovo najnáročnejšie a najviac využívané funkčné časti. [10] [12]

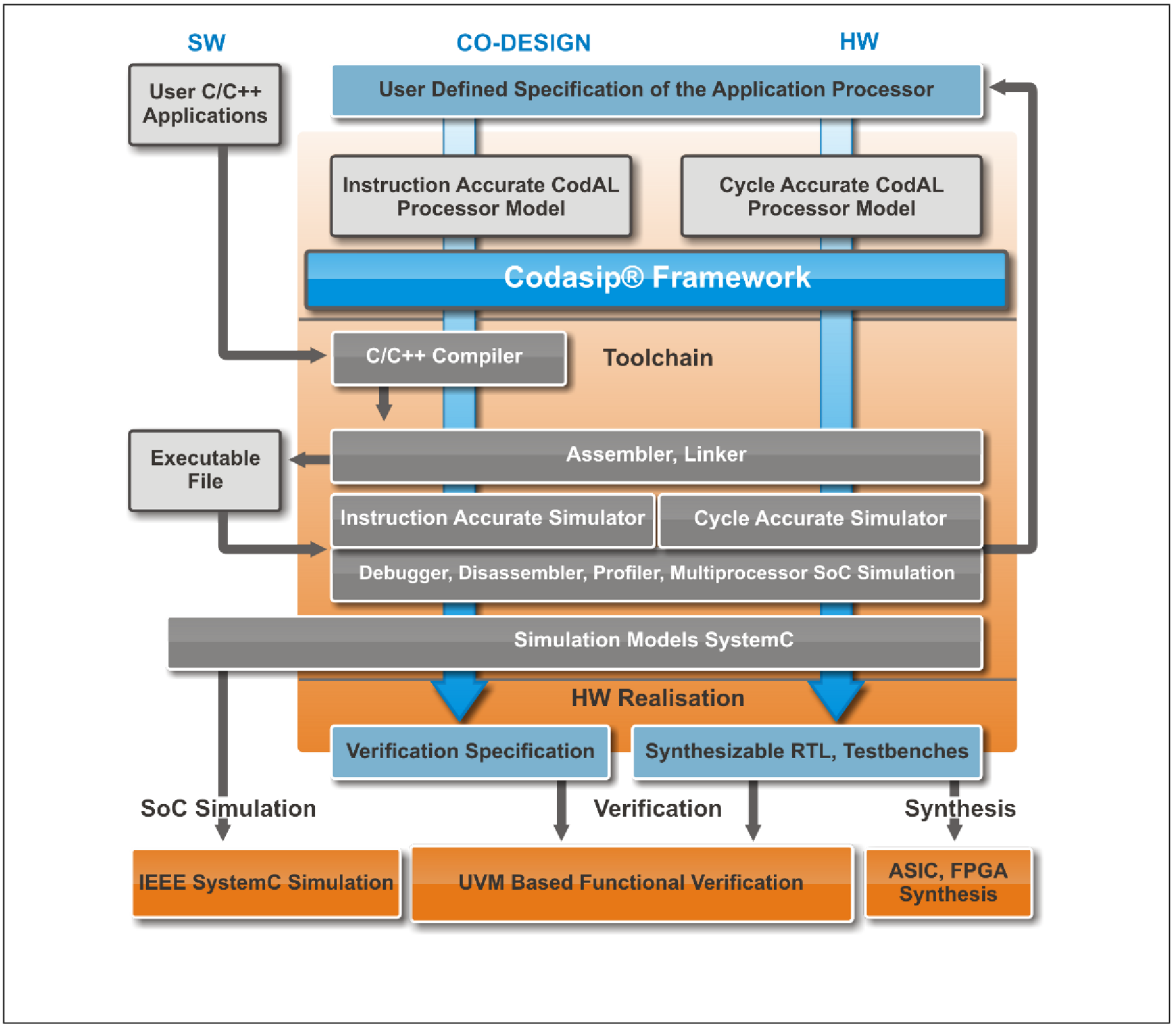
2.1 Codasip Framework

Codasip Framework je pokročilé vývojové prostredie umožňujúce tvorbu vysoko efektívnych procesorov s aplikačne-špecifickou inštrukčnou sadou pre určitú aplikáciu alebo aplikačnú doménu. Codasip Framework je založený na otvorených (anglicky opensource) štandardoch, ktoré poskytujú stabilnú a udržateľnú softvérovú platformu pre výstavbu súboru nástrojov (anglicky toolchain), ktorá je schopná generovať kód pre viaceré cieľové platformy. Codasip Framework automatizuje vytváranie súboru programov slúžiacich na programovanie a simuláciu pre viaceré platformy, generovanie syntetizovateľného návrhu zariadenia a podporu funkčnej verifikácie, ako je možné vidieť na obrázku 1. [16]

Codasip Framework zahŕňa inovatívnu vlastnosť automatizovaného generovania funkčnej verifikácie založenej na metodike UVM (anglicky Universal Verification Metodology). Vďaka tomu môžu návrhári využiť plynulý prechod od špecifikácie procesoru k verifikácii vygenerovaného C/C++ prekladača a syntetizovateľného RTL návrhu. [16]

Na popis procesorov je využívaný jazyk CodAL, ktorý je hierarchickým a vysoko štruktúrovaným popisným jazykom. CodAL slúži na reprezentáciu jadier procesoru na vyššej

úrovni abstrakcie, ktorá je kľúčová pre schopnosť rýchleho a presného prototypovania architektúry. V skorších fázach návrhu je možné vytvoriť popis jadier procesoru na inštrukčnej úrovni, ktorý obsahuje informácie o sémantike inštrukcií. Z tohto popisu je možné generovať prekladač jazykov C/C++ a simulačné nástroje. Po upresnení inštrukčnej sady môže byť vytvorený detailnejší a obšírnejší model v jazyku CodAL popisujúci hardvérovú architektúru procesoru, v angličtine nazývaný cycle-accurate. Kľúčovou vlastnosťou CodAL-u je popis návrhu na inštrukčnej úrovni, ktorý slúži ako referenčný a práve cieľom verifikácie je zabezpečenie jeho ekvivalencie s komplexnejším, cycle-accurate modelom. Pomocou cycle-accurate popisu je teda možné vytvoriť iný popis toho istého procesoru, z ktorého sa dá generovať reprezentácia procesoru v jazyku popisujúcom hardvér (anglicky Hardware Description Language). Tento popis je ďalej syntetizovateľný do skutočného hardvéru, napríklad na poli programovateľných hradiel (anglicky Field-Programmable Gate Array, skrátene FPGA)[16]



Obrázok 1: Codasip Framework [16]

2.2 Procesor Codix RISC

Codix je všestranný procesor navrhnutý s dôrazom na rozšíriteľnosť a jednoduchosť použitia, pričom je vo veľkej miere konfigurovateľný. Vďaka jedinečným možnostiam a schopnostiam prispôsobiteľnej inštrukčnej sady je vhodný pre výpočetné systémy požadujúce vysoký výkon, malé rozmery či nízku spotrebu energie. Procesor Codix RISC má šesť-stupňovú pipeline a 32-bitovú architektúru. Jednou z mnohých výhod procesoru je rýchla optimalizácia na cieľovú skupinu aplikácií pomocou Codasip Framework-u. [16]

Medzi hlavné výhody tohto procesoru patria:

- 32-bitová šírka inštrukcií,
- šesť-stupňová pipeline,
- tridsaťdva 32-bitových registrov na všeobecné použitie s tromi portami na čítanie a jedným portom pre zápis,
- voliteľná vyrovnávacia pamäť pre dáta aj inštrukcie,
- jednoduchá rozšíriteľnosť ako architektúry, tak aj inštrukčnej sady,
- programovanie architektúry pomocou vysokoúrovňových jazykov,
- inštrukčná sada navrhnutá s ohľadom na možnosti LLVM [17] prekladača založená na internej reprezentácii LLVM,
- jadro procesoru optimalizovateľné na nízku spotrebu energie či miesta,
- ponechanie väčšiny hazardov, s výnimkou výpadkov dátovej cache a čakania na pamäťovú zbernicu, ktoré sú v réžii prekladača,
- výkonová porovnateľnosť s architektúrami ARMv7 a Intel686 na úrovni jazyka C,
- voliteľná podpora systému prerušenia. [16]

Vďaka vyššie spomínaným vlastnostiam je procesor Codix RISC vhodný na spracovávanie videa, bezdrátovej komunikácie a iných signálov najmä kvôli jeho schopnosti prispôsobiť sa a optimalizovať jednotlivé algoritmy využívané v kodekoch, bezpečnostných algoritmoch, pri detekcii malvéru a ďalších výpočetne náročných úlohách [16].

Procesor Codix RISC podporuje Linux, Microsoft .NET Micro Framework a iné softvérové aplikácie. Linux je štandardným operačným systémom, ktorý ponúka množstvo voľne dostupného softvéru. Codix je schopný spustiť Linux verzie 3.8.0. s nástrojom BusyBox verzie 1.20, štandardnou knižnicou jazyka C, uClibc, využívajúc iba 3.5 MB pamäte. Vďaka svojim aplikačne-špecifickým inštrukčným rozšíreniam poskytuje vyšší výkon a nižšiu spotrebu pri spustení operačného systému s užívateľsky-špecifickými aplikáciami v porovnaní s bežnými univerzálnymi procesormi využívajúcimi koprocesory a hardvérové akcelerátory. Pre procesor Codix RISC sú dostupné nástroje na podporu programovania a simulácie s modelmi periférnych zariadení v rôznych úrovniach zložitosti popisu za účelom rýchleho ladenia a optimalizácie. [16]

Ďalšou podporovanou hardvérovou abstrakčnou vrstvou je vyššie spomínaný .NET Micro Framework. Patrí k open-source platformám a je navrhnutý pre malé a zdrojovo obme-dzené zariadenia obsahujúce aspoň 256 KB flash pamäte a 64 KB operačnej pamäte, pričom podporuje viacero vlákien (anglicky multi-threading), programovanie aplikácií riadených udalosťami, spracovanie výnimiek apod. [16] [18]

Kapitola 3

Verifikácia hardvérových obvodov

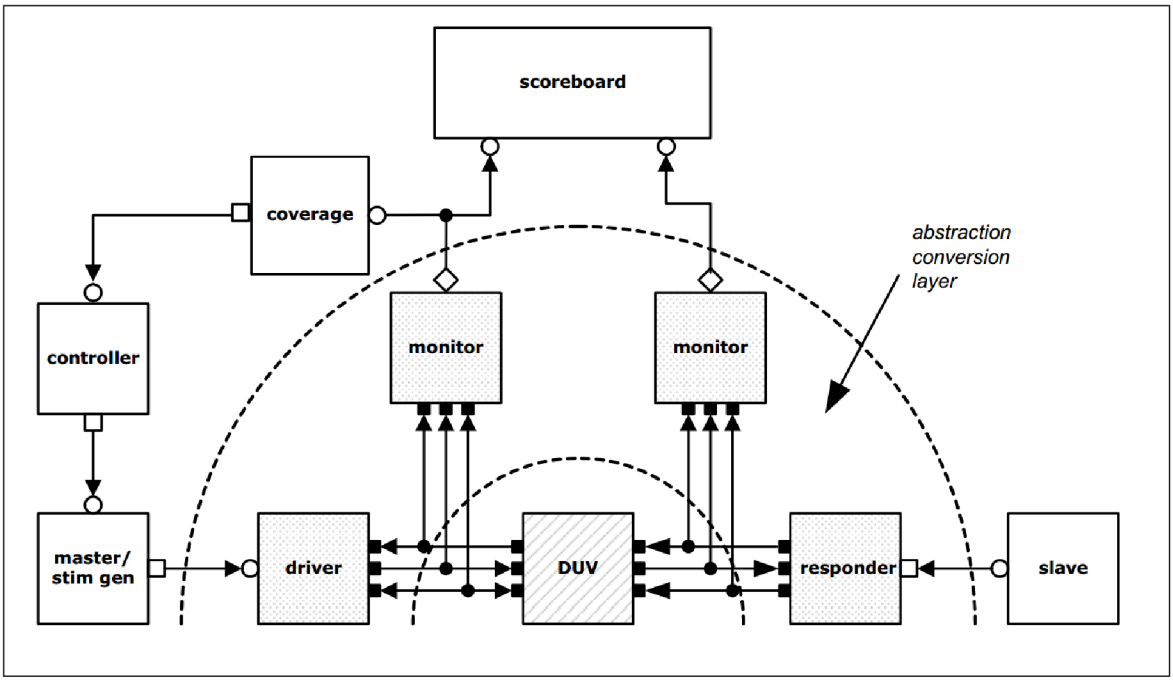
Verifikácia je proces, ktorý overuje, či vyvíjaný produkt spĺňa všetky požiadavky dané špecifikáciou a či sa produkt využíva na zamýšľané účely a vyhovuje užívateľským potrebám. [14] Tradične, inžinieri verifikujú hardvérové jednotky metódou tzv. black-box testovania. Pri tejto metóde sa vytvorí model danej jednotky pomocou jazyka na popis hardvéru (anglicky Hadrware Description Language, skrátene HDL) a následne sa spíše sada testov. Sadu testov v minulosti predstavovali najmä vstupné vektory, ktoré sa aplikovali na testované zariadenia v cykloch. Výstupy jednotlivých cyklov boli priamo porovnávané s referenčnými hodnotami. [6]

3.1 Architektúra verifikačného prostredia

Verifikačné prostredia majú zložitejšiu architektúru a skladajú sa z modulárnych prvkov, ako ilustruje obrázok 2. Tieto prvky sú hierarchicky usporiadané za účelom zjednodušenia konštrukcie, ladenia a údržby komplexných systémov.

V obrázku 2 sú ilustrované nasledovné komponenty [5]:

- `Device under verification (DUV)` je verifikáciou overovaným komponentom.
- `Simulation controller` je hlavným riadiacim prvkom verifikačného prostredia. Uzatvára celý okruh začínajúci generovaním vstupov, pokračujúci driverom, monitorom, scoreboardom a coverage collectorom.
- `Coverage collector` je zodpovedaný za zaznamenávanie pozorovaného chovania v príslušnom modeli pokrytia.
- `Scoreboard` je jedným z verifikačných komponentov pomocou ktorých sa určuje koncová korektnosť verifikovaného zariadenia. Implementuje referenčný model (prediktor), ktorý pre vstupné stimuly pripravuje referenčné výstupy. Tie sú následne porovnávané so skutočnými výstupmi verifikovaného zariadenia (DUV).
- `Stimulus generator` je prvok generujúci vstupné stimuly pre verifikované zariadenie. Obsahuje potrebné algoritmy a zadané obmedzenia.
- `Scenario generator` je obdobou generátoru vstupných stimulov, ktorý vytvára sekvencie stimulov. Ich úlohou je vykonávať špecifickú podrobne popísanú funkciu na verifikovanom zariadení.



Obrázok 2: Testbench [5]

- `Master` je obojsmerný verifikačný komponent ktorý posiela a prijíma odpovede. Master iniciuje aktivitu a môže využívať spätnú väzbu na prispôsobenie ďalších akcií.
- `Slave` je takisto obojsmerný verifikačný komponent, ktorý odpovedá na požiadavky a prijíma odpovede, no na rozdiel od mastera neiniciuje žiadne akcie.
- `Driver` je komponentom konvertujúcim nečasované alebo čiastočne časované transakčné stimuly na časované aktivácie verifikovaného zariadenia na úrovni prepojení.
- `Responder` je podobným komponentom ako driver. Responder sa pripojí na zbernicu a riadi časované aktivácie na úrovni prepojení. Jeho úlohou je hlavne odpovedať na aktivitu na zbernici a nie iniciovať ju.
- `Monitor` je pasívny verifikačný komponent pozorujúci spoje, ktorý konvertuje časovaný popis na prúd transakcií aby ho mohli využívať iné komponenty verifikačného prostredia, predovšetkým scoreboard.

3.2 Formálna verifikácia

Formálna verifikácia využíva matematické metódy k formálnemu popisu systému alebo jeho vlastností a overuje, či je funkčnosť systému v súlade so špecifikáciou. Výsledkom formálnej verifikácie je dôkaz správnosti, ktorý hovorí, že neexistuje žiadny vstup spôsobujúci porušenie sledovanej podmienky, alebo protipríklad dokazujúci, že určitý vstup alebo beh systému vedie k porušeniu podmienky. [20]

Medzi techniky formálnej verifikácie patria:

- `Model checking` - overuje vlastnosti systému úplným preskúmaním jeho stavového priestoru.

- `Statická analýza` - automatická analýza zdrojového kódu, nevyžaduje model systému, používa sa aj pre optimalizáciu a generovanie kódu.
- `Theorem proving` - deduktívna metóda, podobná matematickému dokazovaniu.
- `Equivalence checking` - formálne dokazuje, že dve reprezentácie toho istého obvodu majú rovnaké správanie.
- `SAT Solving` - SAT (satisfiability) reprezentuje NP-úplný problém splniteľnosti booleovských formúl.
- `Assertion-Based Verification (ABV)` - v ďalšom texte uvedenená ako verifikácia založená na formálnych tvrdeniach.

3.3 Verifikácia založená na formálnych tvrdeniach

Verifikácia založená na formálnych tvrdeniach je metodika, pri ktorej návrhári hardvéru a verifikátori využívajú formálne tvrdenia na zachytenie špecifických vlastností systému a následne pomocou simulácie, formálnej verifikácie či emulácie týchto tvrdení overujú korektnosť implementácie systému. [15]

Obrázok 2 zobrazuje typickú verifikáciu systému pomocou tvrdení v transakčnom simulačnom prostredí. Ide o plne rozvinutý, znovupoužiteľný, konfigurovateľný nástroj využívaný na odhaľovanie nie len nesprávneho, ale aj netypického správania. Verifikačné komponenty zodpovedné za vykonanie príslušných akcií detegujú práve takéto správanie. [6]

Vo všeobecnosti je formálne tvrdenie výrokom o zamýšľanom správaní navrhnutého celku, ktorý musí byť vždy platný. Na rozdiel od zdrojového kódu systému v HDL jazyku, formálne tvrdenia nijako nezasahujú v žiadnej forme do správania systému. [6] Formálne tvrdenie je implementáciou vlastnosti, ktorá je vyhodnocovaná a spúšťaná pomocou nástrojov na overovanie korektnosti systému [5]. Inými slovami, formálne tvrdenie definujeme ako tvrdenie pokrývajúce určitú vlastnosť systému, ktorá musí v danom systéme vždy platiť a toto tvrdenie musí byť overené.[21] Štruktúru formálneho tvrdenia zachytáva obrázok 3.

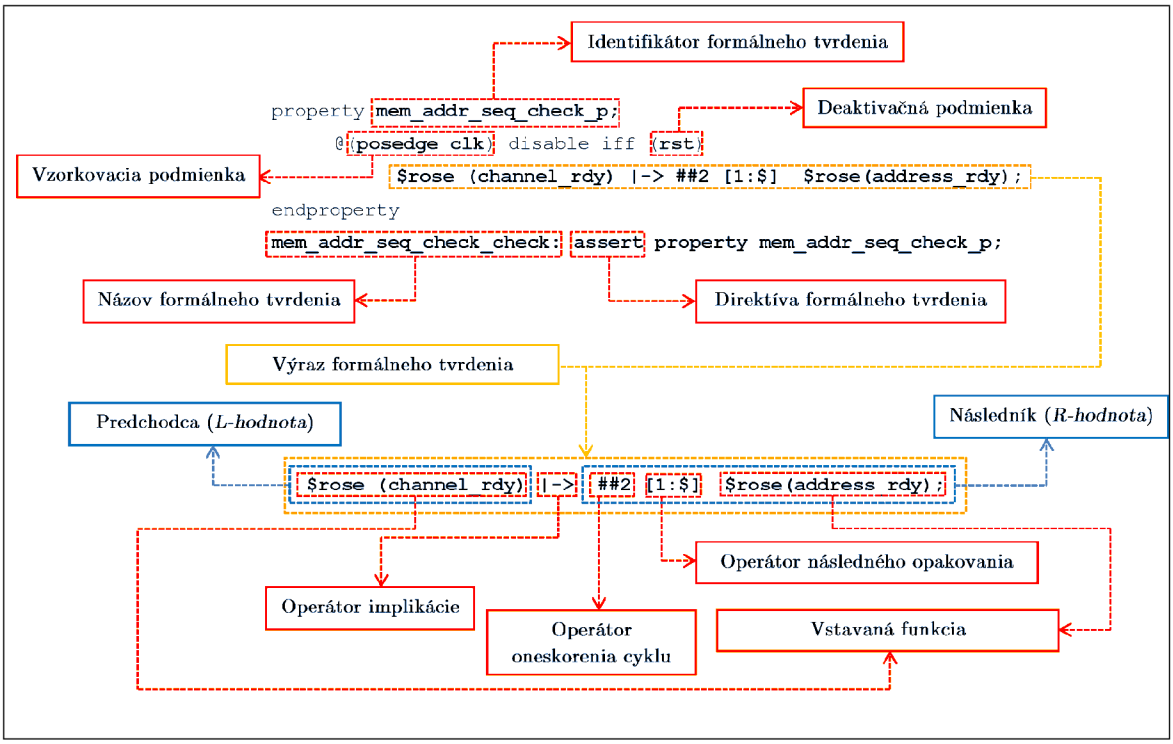
Pomocou jazykov na zápis tvrdení sme schopní zapísať formálne tvrdenia v zrozumiteľnej forme pre dané simulačné či verifikačné prostredie. V praxi sa najviac rozšírili dva jazyky. Prvým je jazyk SystemVerilog Assertions, skrátene SVA, ktorý je podjazykom jazyka SystemVerilog. Jazyku a jeho popisu je venovaná ďalšia podkapitola.

Druhým jazykom je IEEE Property Specification Language, skrátene PSL, slúžiaci na špecifikáciu vlastností, ktoré určujú správanie systému v priebehu času. Jazyk PSL je založený na temporálnej logike a je obohatený o regulárne výrazy.

PSL sa rozšíril v komunite zameranej na verifikáciu hardvéru až po jeho odobrení inštitútom IEEE, kedy sa začal bežne používať na verifikáciu priemyselných produktov. Napriek tomu že PSL je považovaný najmä za jazyk špecifikujúci hardvér, jeho použitie nie je obmedzené len na verifikáciu. Syntaktické rozdiely medzi jazykmi je možné vidieť na príkladoch 3.1 a 3.2. [6]

Okrem jazykov samotných existujú knižnice slúžiace na podporu verifikácie založenej na formálnych tvrdeniach. Medzi najrozšírenejšie patria Open Verification Library, skrátene OVL, ktorá je jazykovo nezávislá a poskytuje rozhranie pre tvorbu tvrdení. Inými používanými sú knižnica SystemVerilog Assertion Library a knižnica CheckerWare Library.

Použitie verifikácie založenej na formálnych tvrdeniach so sebou prináša viaceré výhody. Jednou z najväčších výhod tejto verifikácie je zvýšenie priehľadnosti celého systému. V tra-



Obrázok 3: Štruktúra formálneho tvrdenia [21]

Príklad 3.1 SystemVerilog Assertions [6]

```
always @ (push or pop or cnt or reset_n)
  if(reset_n)
    if({push, pop}==2'b01)
      underflow_check: assert (cnt!=0) else
                          $error(,,underflow error at %m'');
```

Príklad 3.2 IEEE Property Specification Language [6]

```
assert never (reset_n && {push,pop}==2'b10 && cnt==FIFO_depth)
  @(posedge clk);

assert never (reset_n && pop && cnt==0)
  @(posedge clk);
```

dičnom verifikačnom prostredí je vytvorená sada testov, ktorá je aplikovaná na verifikované zariadenie. Vstupom jednotlivých testov je stimul generovaný verifikačným prostredím na základe zadanej sady testov a vstupných podmienok, pričom korektnosť systému sa overuje kontrolou výstupov pre jednotlivé vstupy. Aby bol tento prístup schopný odhaliť chybu, je nutné, aby nastali dve udalosti. Prvou je vyvolanie samotnej chyby, kedy je potrebné systému zadať vstup, ktorý chybu vyvolá. Druhou udalosťou je propagácia tejto chyby do miesta, kde sa výstupy kontrolujú. Pri testovaní môžu nastať prípady, kedy sa pri jednom vstupe nesplnia obe podmienky. Chyba, ktorá v systéme vznikla sa neprejaví vo výstupe, alebo sa prejaví až neskôr pri zadaní iného, inak možno správne fungujúceho vstupu [6].

Naopak, pri verifikácii založenej na formálnych tvrdeniach je chyba reportovaná ihneď. [2]

Za ďalšiu výhodu môžeme považovať skrátenie času potrebného na odladďovanie návrhu, ktorému napomáha detegovanie chýb v blízkosti ich vzniku, a teda presnejšie odhaľovanie príčin problémov v kratšom čase. Bežné postupy verifikácie, ako už bolo spomenuté, kontrolujú výstupy na základe generovaných vstupov a vďaka tomu často nie sú schopné odhaliť zdroj chyby, či dokonca ani čas, kedy chyba nastala, pretože sa chyba prejaví až vo výstupe, častokrát mnoho cyklov po jej vzniku. Na odhalenie podobných problémov je potom potrebné spätne stopovať chybu až k jej vzniku.[6] [4]

Použitím formálnych tvrdení sa okrem sprehľadnenia celého systému ďalej zľahčuje aj integrácia jednotlivých vyvíjaných komponentov. Rozdelením návrhu do celkov a následnou kontrolou dohodnutých vstupov a výstupov jednotlivých dielov je možné presne definovať celok, v ktorom chyba vznikla. Navyše sa dopísaním formálnych tvrdení jasne definuje presné rozhranie medzi celkami. Jednotlivé tvrdenia navyše zostávajú často kontrolované až do finálnej fázy produktu, počas všetkých kontrol a testovaní, aby bolo zaistené, že sa rovnaké chyby opäť nevyskytnú v návrhu aj po ich predošlom overení.[6]

3.4 SystemVerilog Assertions

SystemVerilog umožňuje užívateľovi zapisovať formálne tvrdenia deklaratívne, alebo ich priamo začleniť v procedurálnom kóde. SystemVerilog podporuje dve formy špecifikácie tvrdení: okamžité a konkurentné formálne tvrdenia. [1] [13]

Okamžité formálne tvrdenia sa vyhodnocujú ihneď pri zmene hodnôt premenných vo výraze daného formálneho tvrdenia. Potenciálnou nevýhodou oproti vyhodnocovaniu s taktom hodín je možnosť neplatného porušenia formálneho tvrdenia. Ako prevencia proti takýmto situáciám sa využíva vyhodnocovanie, ktoré počká až všetky aktuálne menené premenné zmenia svoju hodnotu. SystemVerilog zaviedol nový simulačný úsek nazývaný pozorovací región (anglicky observe region), ktorý sa vyhodnocuje po neblokujúcom priradení, čím sa zaisťuje stabilný stav hodnôt premenných. V SystemVerilogu je možné priradiť každému formálnemu tvrdeniu nielen meno, ale aj stupeň závažnosti zobrazený v tabuľke 3.1 .[6]

Stupeň závažnosti	Akcia
Fatálny	Ohlásenie fatálneho stupňa a ukončenie simulácie.
Chyba	Ohlásenie chyby a pokračovanie v simulácii. (prednastavený stupeň)
Varovanie	Ohlásenie varovania, môže byť potlačené.
Informovanie	Ohlásenie všeobecných informácií o tvrdení.

Tabulka 3.1: Stupne závažnosti

Konkurentné formálne tvrdenia popisujú správanie v čase a typicky sa vyhodnocujú s nábežnou alebo zostupnou hranou hodinového signálu, voči ktorému je synchronizovaný celý systém. Jedným taktom hodinového signálu v texte ďalej rozumieme časový úsek medzi dvomi nábežnými hranami, začínajúci nábežnou hranou. Rovnako ako pri okamžitých formálnych tvrdeniach, aj pri konkurentných zaviedol SystemVerilog nový simulačný časový úsek tzv. predčasový región. Pri konkurentných formálnych tvrdeniach je možné definovať vlastnosť, ktorá môže byť následne využitá v tvrdení, funkčnom pokrytí, či ako obmedzenie, kedy daná vlastnosť limituje vstupné hodnoty. Vlastnostiam je možné priradiť aj meno a argumenty, vďaka čomu sa z vlastností stávajú veľmi dobre znovupoužiteľné vzory a predlohy.

Následne je možné sa na ne odvolávať menom s použitím parametrov, či dočasne obmedziť ich funkčnosť (príklad 3.3). Jeden z možných postupov pri verifikácii je použitie vlastnosti ako formálneho tvrdenia, ktorého prípadné zlyhanie sa hlási simulačnému prostrediu. Druhý spôsob môžeme vnímať ako komplement prvého, kedy zaistíme, že výskyt danej vlastnosti bude zaznamenaný. [6]

Príklad 3.3 Deklarácia vzájomnej výlučnosti

```
property mutex (clk, reset_n, a, b);
    @(posedge clk) dissable iff (reset_n) (!(a & b));
endproperty
```

Formálne výrazy v jazyku SystemVerilog Assertions majú svoju hierarchiu, ktorá je zobrazená na obrázku 4. Základným stavebným kameňom je booleovský výraz špecifikujúci udalosti v jednom hodinovom takte, pričom sa výraz vyhodnocuje ako pravdivý či nepravdivý vždy s nástupnou či zostupnou hranou.[21]

Vzťahy medzi booleovskými výrazmi v simulačnom čase definujú sekvencie. Sekvencie sú konečné série booleových udalostí, kde každý výraz reprezentuje lineárny vývoj v čase, teda sekvencia popisuje špecifické správanie [6]. Spolu s regulárnymi výrazmi to rozširuje naše možnosti o podmienky, kedy musia dané výrazy platiť. Pomocou rôznych operátorov je možné špecifikovať ďalšie možnosti ako sú opakovanie, výskyt paralelných, voliteľných či podmienených vetiev. Navyše je možné sledovať aj po sebe nenasledujúce udalosti, napríklad výskyt rovnakej udalosti, avšak medzi udalosťami môže nastať istý, dokonca možno aj bližšie špecifikovaný počet cyklov.

Vlastnosť je kolekcia logických a temporálnych vzťahov medzi a nad logickými vý-razmi, sekvenčnými výrazmi a inými vlastnosťami, agregovanie ktorých reprezentuje súhrn správaní. [6] Vlastnosti umožňujú rôzne využitie výrazov a sekvencií ako je podmieňovanie, rušenie a povolovanie.

Vlastnosťou v kontexte verifikácie založenej na formálnych tvrdeniach je výrok o očakávanom správaní. [5] Príkladom môže byť výrok, že pravdivosť logických hodnôt premenných s menami `bool1` a `bool2` je vzájomne výlučná. Túto vlastnosť môžeme vyžadovať od časti nášho návrhu a tvrdiť, že táto vlastnosť bude splnená počas celého procesu verifikácie.

Pomocou direktív určujeme, ako budú dané vlastnosti, sekvencie a booleovské výrazy využité v danom systéme, pričom formálne tvrdenie sa skladá z direktívy a vlastnosti. Direktíva `assert` určuje vždy platnú vlastnosť či podmienku, `cover` definuje požiadavku na sledovanie výskytu vlastnosti v systéme za účelom statického vyhodnocovania a `assume` obmedzuje vstupné hodnoty verifikovanej jednotky.

Verifikácia založená na formálnych tvrdeniach kladie veľký dôraz na znovupoužiteľnosť a je aplikovateľná vo všetkých fázach vývoja hardvéru. Vo fáze návrhu ide o o automatickú kontrolu formálnych tvrdení a využitie knižníc či vlastných vzorov. Pomocou automatickej kontroly sa odlaďujú najmä chyby aritmetické, chyby v konečných automatoch a chyby v prepojení registrov. [21]

Ďalšou aplikáciou ABV je statická formálna verifikácia, kedy sa *dokazuje* platnosť formálnych tvrdení a overujú sa kľúčové vlastnosti obvodov voči špecifikácii. Často sa na tieto účely využívajú model-checking alebo bounded model-checking metódy.

Pomocou funkčnej verifikácie je možné simulovať beh zariadenia a overovať platnosť formálnych tvrdení. Výhodou simulácie je okamžité upozornenie na porušenie podmienky a získanie informácii o pokrytí sledovaných podmienok.

Obrázok 4: Hierarchia formálnych výrazov v jazyku SVA [21]

Kombináciou funkčnej a statickej formálnej verifikácie sa dostávame k dynamickej formálnej verifikácii, ktorá vykonáva lokálnu formálnu analýzu v okolí zaujímavých stavov.

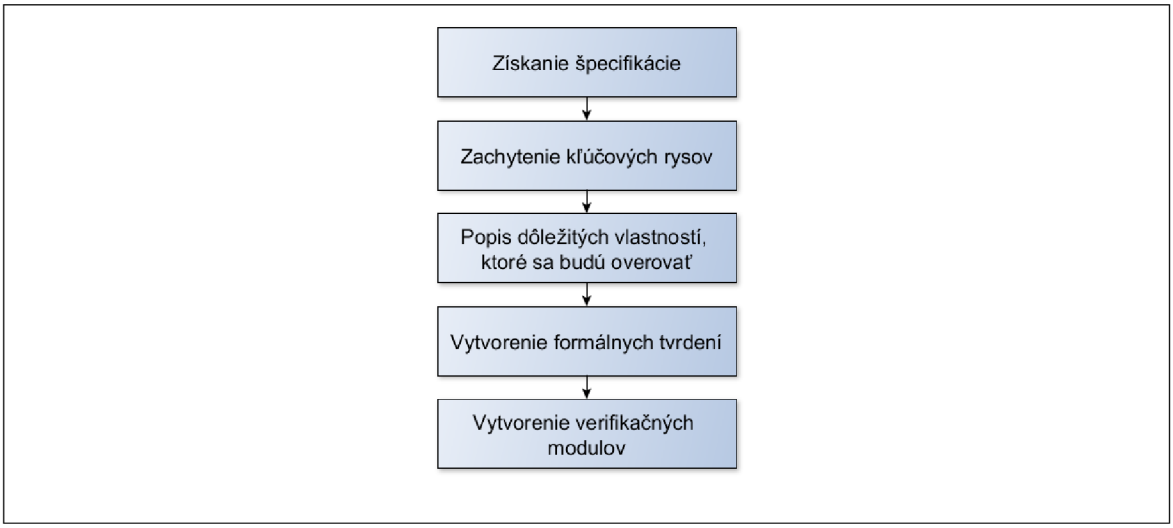
Po výrobe zariadenia je možné monitorovať správanie zariadenia pripojením samostatnej hardvérovej súčiastky vytvorenej syntézou formálnych tvrdení. [21]

Kapitola 4

Verifikácia založená na formálnych tvrdeniach pre procesory typu ASIP

4.1 Verifikačný proces

Verifikačný proces a sled jeho krokov je znázornený na obrázku 5. Prvým krokom v procese verifikácie založenej na formálnych tvrdeniach je získanie špecifikácie, blokového diagramu, popisu signálov a komunikačného protokolu. Získané špecifikácie a poznatky budú využité pri tvorbe formálnych tvrdení a sledovaní ich pokrytia. Druhý krok spočíva vo vytvorení zhrňujúceho popisu správania systému, v ktorom sú zachytené kľúčové charakteristiky návrhu. Nie je dôležitá obšírnosť tohto popisu, no je nevyhnutné aby zachytával všetky podstatné rysy a funkcie. V kroku tri sa vytvorí slovný popis všetkých dôležitých vlastností návrhu, ktoré budeme overovať. Štvrtý krok vytvára formálne tvrdenia v jazyku SystemVerilog na základe slovného popisu z predošlého kroku a získava ich pokrytie. V kroku päť je vytvorený súbor znovupoužiteľných verifikačných komponentov, ktorý vznikol zapúzdrením formálnych tvrdení do modulov a rozhraní. [5]



Obrázok 5: Verifikačný proces [5]

4.2 Komponenty

Táto podkapitola je venovaná popisu hlavných komponentov, ktoré tvoria základ procesorov s aplikačne-špecifickou inštrukčnou sadou. Ich korektnosť je preto potrebné overiť v procese verifikácie. [9]

4.2.1 Fronta, Zásobník, Konečný automat, Multiplexor, Pamäť

Jedným z najpoužívanejších elementov je fronta, tzv. FIFO (anglicky First-In First-Out), ktorá sa využíva ako vyrovnávacia pamäť medzi asynchrónnymi prvkami spracovávajúcimi dáta. Fronta je najčastejšie overovaná na pretečenie, podtečenie, chybné dáta či interný stav fronty. Príklad 4.1 znázorňuje formálne tvrdenie, ktoré overuje podmienku pretečenia. Časť „assert never“ určuje, že podmienka, ktorá nasleduje, nesmie byť nikdy splnená. Príklad ukazuje dve varianty zápisu vyjadrujúceho stav plnej fronty, každý zobrazený v samostatnom riadku. Fronta je plná, ak je signál `full` v stave logickej jednotky alebo sa počítadlo prvkov `cnt` rovná veľkosti fronty `FIFO_Depth`. Pretečenie nastáva ak je fronta plná, a zároveň sa do nej zapisuje a nie je z nej čítané. V tomto prípade sa signál `push` rovná logickej jednotke a signál `!pop` je v logickej nule. Opačným príkladom formálneho tvrdenia je kontrola podtečenia, kedy je fronta prázdna pričom signály čítania a zápisu sú v stave logickej jednotky, respektíve nuly. Obdobným, taktiež bežne sa vyskytujúcim komponentom, je zásobník tzv. LIFO (anglicky Last-In First-Out).

Príklad 4.1 Formálne tvrdenie v jazyku SystemVerilog overujúce, že fronta nepretečie [6]

```
assert never (cnt == FIFO_depth & push & !pop);
assert never (full & push & !pop);
```

Konečné automaty sú jedným z najvhodnejších komponentov pre verifikáciu založenú na formálnych tvrdeniach. Zároveň reprezentujú ďalšiu skupinu jednoduchších komponentov slúžiacich ako základné stavebné prvky. Pomocou verifikácie je v nich možné kontrolovať a overovať ilegálne stavy, kombinácie prechodov a mnoho iných vlastností. Príklad 4.2 ukazuje jednoduchú kontrolu legálnych stavov konečného automatu. V každom takte sa kontroluje či aktuálny stav patrí do množiny platných stavov { IDLE, READ, WRITE, START, END }. V tomto príklade je využítá podmienka obmedzujúca platnosť formálného tvdenia. V prípade, ak sa systém nachádza v stave `reset`, teda signál `rst_n` je v nule (reset aktívny v nule), sa podmienka ďalej neoveruje a je považovaná za splnenú. Takéto obmedzovanie je v praxi časté najmä kvôli okrajovým podmienkam a situáciám, ktoré nastávajú pri resete, na začiatku a na konci simulácie. Okrem toho sa v praxi často využíva aj tzv. vypínač (viď príklad 4.3), teda premenná či signál slúžiaci na vypnutie všetkých či určitej skupiny formálnych tvrdení, ktorých overovanie nie je požadované. Toto formálne tvrdenie (príklad 4.3) zabezpečuje, že signály `req` a `abort` nebudú v stave logickej jednotky v prípade, ak sa automat nachádza vo vymenovaných stavoch.

V dnešnej dobe nenájdeme takmer žiadne hardvérové komponenty, ktoré by neobsahovali *multiplexor* či už sa jedná o kóder, dekodér alebo iný typ. Typicky sa pri multiplexoroch kontroluje počet aktívnych výstupov, správnosť dát, povolené kombinácie a pod. Príkladom verifikácie multiplexoru je kontrola aktivity aspoň jedného výstupu ako znázorňuje príklad 4.4, kde signál `select` reprezentuje výber vstupov prepojených na výstup. Každý vstupný signál je reprezentovaný jedným bitom v signále `select` a hodnota daného bitu určuje jeho

Príklad 4.2 Formálne tvrdenie v jazyku SystemVerilog overujúce stavy konečného automatu [6]

```
property legal_states;
  @(posedge clk) disable iff !rst_n
    state inside {'IDLE, 'READ, 'WRITE, 'START, 'END };
endproperty
assert property (legal_states);
```

Príklad 4.3 Formálne tvrdenie s využitím tzv. vypínačov

```
property legal_states;
  @(posedge clk) disable iff (switch_local or switch_global or !rst_n)
    not ((state inside {'READ, 'READ1 }) & (req | abort))
endproperty
assert property (legal_states);
```

prepojenie na výstup. Formálne tvrdenie overuje, či v každom takte signál `select` obsahuje aspoň jeden bit s hodnotou 1, resp. celková hodnota nie je rovná nule (4'b0).

Príklad 4.4 Formálne tvrdenie v jazyku SystemVerilog overujúce výber multiplexoru [6]

```
property active_select;
  @(posedge clk) disable iff 1rst_n
      not (select == 4'b0);
endproperty
assert property (active_select);
```

Pamäť je ďalším zo základných komponentov, ktoré sú vhodným kandidátom pre verifikáciu založenú na formálnych tvrdeniach, ktorými je možné kontrolovať platnosť adresy, dát, ako aj operácie čítania a zápisu. Spomínanú kontrolu neplatnosti zadanej adresy je možné vidieť v príklade 4.5, ktorá s každým výskytom hodinového signálu `clk` overuje nasledovnú podmienku. Signály `ce_n` (clock enable - povolenie akcie v danom takte) a aspoň jeden z dvojice `rd_n` a `we_n` (povolenie čítania a zápisu) musia byť v stave logickej jednotky, a zároveň v danom takte nesmie byť hodnota signálu `address` neznáma.

Príklad 4.5 Formálne tvrdenie v jazyku SystemVerilog overujúce správnosť adresy [6]

```
property forbidden_address;
  @(posedge clk) not ( ce_n & ( rd_n \  we_n ) ##0  $isunknown(address));
endproperty
assert property (forbidden_address)
    else $error(Unknown address);
```

4.2.2 Rozhrania a zbernice

Rozhrania a zbernice slúžia vo všeobecnosti na prepojenie prvkov systému. Na to, aby prvky medzi sebou vedeli navzájom komunikovať je potrebné, aby sa správali podľa určených pra-

vidiel. Z toho plynie množstvo potenciálnych zlyhaní a chýb, ktoré je možné často efektívne a jednoducho overiť pomocou formálnych tvrdení. Medzi bežne overované vlastnosti patria následnosti vzájomne podmienených signálov ako napríklad:

- po signále `reset` musí byť dátový signál `SBD` a hodinový signál `SBC` uvedený do stavu logickej jednotky (príklad 4.6),
- nový príkaz `start` nesmie byť vykonaný, kým nepríde ukončovací príkaz `finish` (príklad 4.7),
- signál `end` môže byť zaslaný až po signále `start` (príklad 4.8).

Príklad 4.6 Formálne tvrdenie v jazyku SystemVerilog overujúce stav logickej jednotky signálov po resete [5]

```
property reset;
  @(posedge monitor.sclk)
    $fell(monitor.reset) |=>
        (monitor.SBC == 1'b1 &monitor.SBD == 1'b1);
endproperty
assert property (reset)
```

V príkladoch 4.7 a 4.8 je využitá rovnaká konštrukcia „throughout“ vyjadrujúca, že podmienka pred týmto kľúčovým slovom po počiatočnom splnení ostane platná počas celého časového intervalu určeného podmienkou za slovom throughout. Konkrétne „throughout“ v príkladoch overuje, že po prijatí príkazu `start` môže prísť ľubovoľný signál rôzny od `start`, inými slovami, udalosť štart nesmie znovu nastať, až pokiaľ sa nevyskytne príkaz `finish`. Analogicky v príklade 4.8 zaručuje, že príkaz `finish` nepríde bez predchádzajúceho príkazu `start`.

Príklad 4.7 Formálne tvrdenie v jazyku SystemVerilog overujúce správnosť sledu začínajúceho a ukončovacieho signálu [5]

```
property double-start;
  @(posedge monitor.sclk) disable iff (monitor.rst)
    monitor.start |=>
       ( monitor.start throughout monitor.finish[->1]);
endproperty
assert property (double-start)
```

Príklad 4.9 znázorňuje kontrolu špecifického sledu signálov v transakcii. Tá pozostáva zo signálu `start`, za ktorým nasleduje sedem bitov adresy, signál `read`/`write`, signál `ACK`, sekvencia ôsmich prenášaných bitov, signál `ACK` a signál `finish`. Zaznamenanie signálu `start` znamená, že na platnosť formálneho tvrdenia uvedeného v príklade je potrebné, aby nasledovali aspoň dva osem-bitové cykly pozostávajúce buď zo siedmych bitov adresy a jedného bitu signálu `read`/`write`, alebo ôsmich bitov prenášaných dát. Všetky cykly sú ukončené signálom `ACK`. Týchto `n` cyklov musí byť nasledovaných signálom `finish` oneskoreným o jeden takt, ktorý celý sled ukončuje.

Príklad 4.8 Formálne tvrdenie v jazyku SystemVerilog overujúce správnosť sledu začínajúceho a ukončovacieho signálu [5]

```
property finish-before-start;
  @(posedge monitor.sclk) disable iff (monitor.rst)
    monitor.finish |=>
       ( monitor.finish throughout monitor.start[->1]);
endproperty
assert property (finish-before-start)
```

Príklad 4.9 Formálne tvrdenie v jazyku SystemVerilog overujúce následnosť signálov na zbernici [5]

```
property valid_transfer_size;
  @(posedge monitor.sclk) disable iff (monitor.rst)
// priebeh minimálne dvoch cyklov pre adresu a nasledované dáta
    monitor.start |=>
        ($rose(monitor.SBC)[=8] ##1 monitor.ack)[*2:$] ##1 (monitor.finish);
endproperty
assert property (valid_transfer_size)
```

4.2.3 Arbitery

Arbitery sú neodmysliteľné komponenty systémov zdieľajúcich svoje zdroje ako sú prístup k zbernici či právo na zápis. Pri rozhodovaní využívajú rôzne arbitračné schémy popisujúce správanie arbitra, a preto sú obzvlášť vhodné na verifikáciu pomocou formálnych tvrdení, ktoré toto správanie dokážu popísať [5]. Často využívanými schémami sú najmä:

- Férová arbitrácia znázornená na príklade 4.10 a popísaná nižšie spočíva v rovnomernom prideľovaní požadovaných zdieľaných zdrojov. Každý žiadateľ má zaručnené, že jeho požiadavka bude spracovaná najneskôr v `n` taktoch, kde `n` je počet možných žiadateľov.
- Špecifické arbitračné schémy:
  - Vzájomná výlučnosť uvedená v príklade 4.12 obmedzuje počet prístupov ku zdieľanému zdroju na maximálne jedného žiadateľa.
  - Arbitrácia s minimálnou latenciou vyžaduje minimálne oneskorenie medzi žiadosťou o pridelenie zdieľaného zdroja a pridelením prístupu k danému zdroju, napríklad (`$rose(req[i]) ##[5:$] gnt[i]`).
- Prioritné arbitračné schémy:
  - Arbitrácia s pevnou prioritou (príklad 4.11) má fixne zadané priority vstupov pričom požiadavku s vyššou prioritou musí spracovať prednostne.
  - Arbitrácia s dynamickou prioritou je taká, ktorej priorita sa často mení v závislosti od urgentnosti požiadavky, hrozieb (strata a poškodenie dát) a od odhadovaného dopadu akcií.

– Arbitrácia s prioritou využívajúcou kredity prideľuje jednotlivým žiadateľom určitý počet kreditov, ktoré sa po čase obnovujú. Pri požiadavke na pridelenie zdieľaného zdroja sa kredity odčítajú a bez nich nie je možné získať prístup k požadovaným zdrojom.

Implementáciu férovej arbitrácie pre ôsmich klientov môžeme vidieť na príklade 4.10, ktorý využíva generovanie assert príkazov pomocou cyklov a konštrukcie „generate“. Táto konštrukcia pomocou vnorených cyklov vygeneruje zo všeobecného parametrizovateľného formálneho tvrdenia všetky požadované varianty. Samotné formálne tvrdenie verifikuje prípady, kedy komponent „i“ žiada o prístup ku zdrojom, pričom pre všetky ostatné komponenty (pre každú kombináciu je dané formálne tvrdenie vygenerované) platí, že im nesmie byť pridelený požadovaný prístup dvakrát po sebe bez toho, aby bol predtým pridelený pôvodnému žiadateľovi „i“. V konečnom dôsledku to znamená férovú arbitráciu pretože všetkým žiadateľom bude pridelený prístup v stanovenom čase a s rovnakými podmienkami.

Príklad 4.10 Férová arbitrácia pre 8 klientov [5]

```
property no_req_i_two_gnt_j (i, j);
  @(posedge clk) disable iff (rst)
    req[i] ##1 (!gnt[i] throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i=0; i<=7; i++) begin
      for (j=0; j<=7; j++) begin
        assert property (no_req_i_two_gnt_j);
      end
    end
  end
endgenerate
```

Rovnako aj v príklade 4.11 slúži generovanie formálnych tvrdení všetkých kombinácií na uľahčenie, sprehľadnenie a zjednodušenie celého systému. Overované formálne tvrdenie je hneď na začiatku obmedzené na prípady, kedy je číslo `i` menšie ako číslo `j`. Toto obmedzenie je možné aplikovať už pri generovaní formálnych tvrdení. Pri prijatí požiadavky s vyššou prioritou nesmie nastať situácia, kedy povolenie na prístup s vyššou prioritou nenastane predtým ako je udelené povolenie požiadavke s nižšou prioritou (jeho prvý výskyt). V prípade, že taká situácia nastane, bude formálne tvrdenie vyhodnotené ako neplatné kvôli splneniu podmienky `|-> 0`.

Ďalším typickým príkladom pri overovaní správnej arbitrácie je zabezpečenie vzájomnej výlučnosti dvoch či viacerých signálov. V príklade 4.12 je uvedená implementácia tejto vlastnosti pre dva signály. Vzájomná výlučnosť znamená exkluzivitu prístupu ku zdieľanému zdroju, teda viacero žiadateľov nesmie súčasne k tomuto zdroju pristupovať. Príklad ukazuje formálne tvrdenie, ktoré zaručuje výlučný prístup pre dvoch žiadateľov, konkrétne, nesmie nastať situácia, kedy obaja žiadatelia získajú prístup ku zdieľanému zdroju, čiže `!(gnt[0] & gnt[1])`.

Príklad 4.11 Arbitrácia s pevnou prioritou pre osem klientov [5]

```
// priorita i > j ak i < j
property fixed_priority_i_j (i, j);
  @(posedge clk) disable iff (rst)
    (i < j) && $rose (req[i])
    ##1 (~gnt[i] throughout (gnt[j])[->1]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i=0; i<=7; i++) begin
      for (j=0; j<=7; j++) begin
        assert property (fixed_priority_i_j);
      end
    end
  end
endgenerate
```

Príklad 4.12 Vzájomne výlučné povolenia [5]

```
property mutex_gnt;
  @(posedge clk) disable iff (rst)
    !(gnt[0] & gnt[1]);
endproperty
assert property (mutex_gnt);
```

Kapitola 5

Súvisiaca práca

Cieľom kapitoly je podať čitateľovi prehľad o súčasnom využívaní verifikácie založenej na formálnych tvrdeniach, o možnostiach jej uplatnenia v praxi a celkovom prínose v oblasti návrhu a vývoja procesorov.

5.1 Verifikácia sieťového procesoru založená na formálnych tvrdeniach

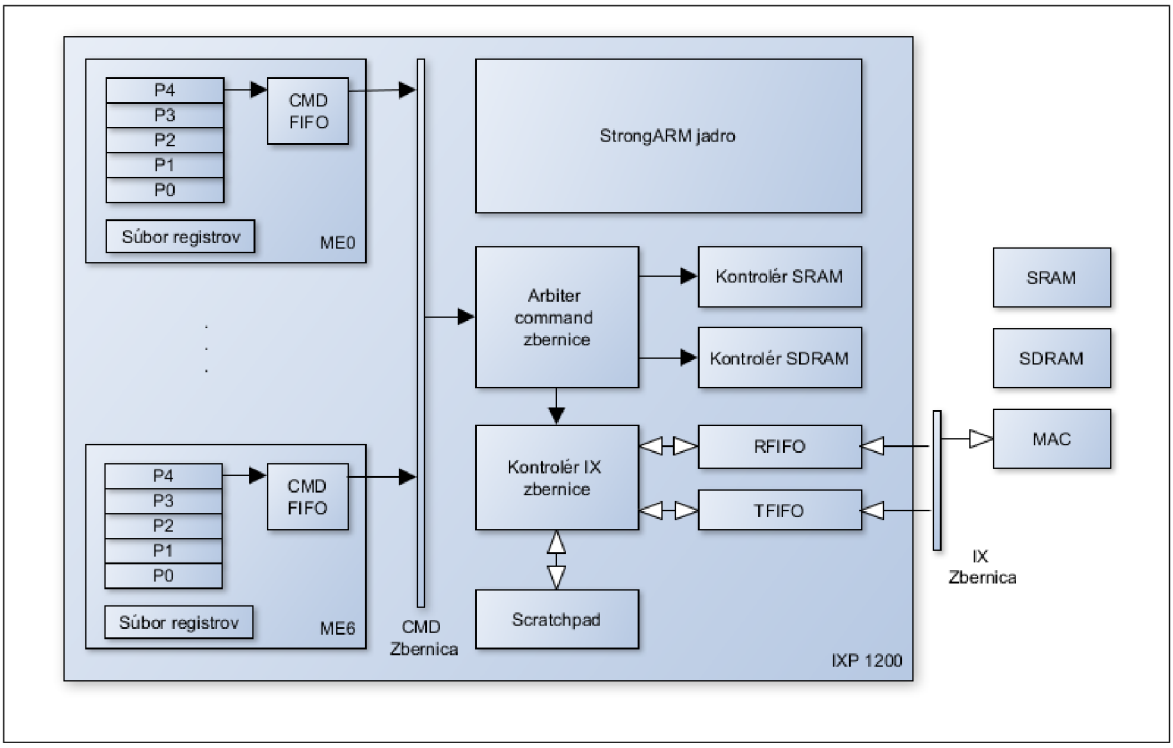
V práci [3] je cieľom autorov overiť správnu funkčnosť triedy bežných sieťových procesorov. Ako referenčný model, resp. zástupcu danej skupiny, zvolili Intel IXP1200, a to najmä vďaka jeho popularite a rozšírenosti.

Autori v práci využívajú dva druhy jazykov. Prvým druhom sú jazyky založené na lineárnej temporálnej logike (anglicky Linear Temporal Logic, skrátene LTL), ktorých úlohou je v prvom rade overovať funkčné formálne tvrdenia. Pomocou jazyka z tejto kategórie sa overujú primárne funkčné vlastnosti napr. vzájomná výlučnosť či vyhladovenie. Autori k tomuto účelu zvolili jazyk Sugar2.0.

Okrem jazyka založeného na LTL sa v práci využíva aj logika obmedzení (anglicky Logic of Constraints, skrátene LoC). Pomocou nej je možné zapísať výrazy, ktorých výsledkom sú rôzne kvantitatívne ukazovatele, či už výkonové ako sú rýchlosť a latencia, alebo vlastnosti transakčnej úrovne ako konzistencia vstupno/výstupných dát.

Intel IXP1200, ktorého architektúra je zobrazená na obrázku 6 je procesor pozostávajúci zo StrongARM jadra, šiestich RISC mikroprocesorov, ktorých inštrukčná sada je optimalizovaná na prácu so sieťovými paketmi. Každý z mikroprocesorov podporuje až štyri vlákna. Pomocou zbernice komunikuje s kontrolérom statickej pamäte s náhodným prístupom (anglicky Static Random Access Memory, skrátene SRAM) a kontrolérom synchrónnej dynamickej pamäte s náhodným prístupom (anglicky Synchronous Dynamic Random Access Memory, skrátene SDRAM), obsahuje vlastné plánovacie jednotky, aritmeticko-logickú jednotku (anglicky Arithmetic Logic Unit, skrátene ALU) a pipeline, pričom každé vlákno má samostatnú sadu registrov. Pipeline každého mikroprocesoru má päť stupňov a každý mikroprocesor má svoj dvojcestný zásobník príkazov, ktoré sa z neho následne posielajú na zbernicu. Vytvorený model procesoru je parametrizovateľný či už sa jedná o počet vlákien v mikroprocesore, počet mikroprocesorov, ich úlohu alebo veľkosť FIFO.

Logic of Constraints je formalizmus navrhnutý na zhodnocovanie vykonaných postupov z kvantitatívneho hľadiska a rovnako je veľmi vhodný na analýzu výsledkov z vyšších vrstiev. Obsahuje výrazy a operátory zo sekvenčnej logiky s možnosťou ich kvantitatív-



Obrázok 6: Architektúra procesoru IXP1200

neho overovania pomocou tvrdení. Príklad 5.1 ukazuje zápis podmienky, ktorá stanovuje, že medzi `n`-tou a `n+100` udalosťou nesmie prebehnúť viac ako 1000 časových intervalov.

Príklad 5.1 Zápis podmienky v LOC

```
cycle(pipeline [n + 100]) - cycle(pipeline[n]) = 1000
```

Autori [3] sa zamerali na tri hlavné kategórie formálnych tvrdení, podľa ktorých bola práca rozdelená na viacero fáz. Prvou kategóriou je overovanie samotného modelu, druhou je funkčná verifikácia a treťou výkonnostné merania. Verifikácia samotného modelu spočíva v overovaní, či model verne reflektuje vlastnosti reálneho procesoru. Základom je porovnávanie výsledkov simulácii s využitím modelu s rovnakými parametrami ako má procesor IXP1200. Simulácia nevyhodnocuje stav všetkých komponentov v každom takte hodinového signálu, ale využíva procesorové udalosti ako sú:

- inštrukcia bola priradená do pipeline,
- inštrukcia opustila pipeline,
- pamäťový príkaz bol vložený/vybraný do/z FIFO,
- signály sú generované alebo prijímané funkčnými jednotkami.

V prvej fáze sa autori zamerali na verifikáciu správnosti modelu po funkčnej a výkonovej stránke. Na vytvorenom modeli aj reálnom procesore spustili testovacie sady zamerané na vyhľadávanie v smerovacích tabuľkách, ktoré vyžadujú správne čítanie informácií z SRAM jednotky. Príklad 5.2 vyjadruje, že pre každý prístup do SRAM musí byť odkazovaná adresa

a výsledná prečítaná hodnota rovnaká a všetky požiadavky na SRAM musia byť spracovávané v rovnakom poradí pre model aj fyzické zariadenie.

Príklad 5.2 Overovanie správnej funkčnosti

```
addr(sram enq[i]) = addr(sram enq IXP[i]) &&
data(sram done[i]) = data(sram done IXP[i])
```

Jedným z výkonových ukazateľov, ktoré bolo potrebné overiť je podobnosť správania a výkonu inštrukčnej pipeline. Konkrétnejšie, inštrukcie na reálnom aj modelovom procesore sú spúšťané v rovnakom poradí a časový rozdiel medzi spustením inštrukcie na procesore a modele nie je viac ako n cyklov. Prvá časť LOC formule (Príklad 5.3) zaručuje zhodnosť poradia. Druhá hovorí, že časy behu sú v určitom pomere, ktorý sa vzhľadom na prebiehajúcu simuláciu akumuluje a odráža taktiež rozdielne časy prípravy behu testovacej sady. Rovnakým spôsobom je možné overiť aj jednotky SRAM a SDRAM.

Príklad 5.3 Porovnanie vzájomného oneskorenia

```
(pc(pipeline[i]) = pc(pipeline)) &&
abs(cycle(pipeline[i]) - cycle(pipeline_IXP[i])) =< A~x i + b
```

Vo fáze funkčnej verifikácie boli overované vlastnosti ako napr:

- každá požiadavka na SRAM jednotku musí byť spracovaná do 300 výskytov obdobných signálov (príklad 5.4),
- požiadavka vložená do fronty z nej musí byť odstránená predtým, než môže byť ukončená (príklad 5.5),
- pamäťová referencia požiadavky vo fronte musí ostať nemenná počas jej celého spracovania (príklad 5.6).

Príklad 5.4 Časové obmedzenie trvania spracovania

```
always(m0_t0_sram_enq -> next_e[1:300](m0_t0_sram_done))
```

Príklad 5.5 Správne poradie udalostí

```
always(m0_t1_sram_enq -> !m0_t1_sram_done until m0_t1_sram_deq)
```

Príklad 5.6 Pamäťová konzistencia

```
addr(m1_t2_sram_enq[i]) = addr(m1_t2_sram_done[i])
```

Z hľadiska výkonnosti bolo cieľom odhaliť možnosti daného celku, otestovať rôzne konfigurácie modelovaného systému a prípadne nájsť architektúru, ktorá by v rôznych oblastiach prekonávala pôvodný modelovaný systém.

Jednou z hlavných činností daného procesoru je vyhľadávanie IP adries. Z toho autori [3] usúdili, že kritickou časťou z pohľadu výkonu je latencia medzi zadaním a potvrdením požiadavky na prístup do SRAM (príklad 5.7) a celková doba ktorú mikroprocesor (jedno z jeho vlákien) potrebuje na vyhľadanie IP adresy (príklad 5.8).

Príklad 5.7 Latencia SRAM požiadavky

```
cycle(m2_t0_ip_sram_done[i]) - cycle(m2_t0_sram_enq[i]) =< 12
```

Príklad 5.8 Latencia vyhľadania IP adresy

```
cycle(m0_t0_ip_lookup_done[i]) - cycle(m0_t0_lookup_start[i]) =< 12
```

Ďalším významným ukazateľom overovaným v práci [3] je priepustnosť či už sa jedná o počet vykonaných požiadaviek (príklad 5.9), alebo o možstvo preposlaných IP paketov (príklad 5.10).

Príklad 5.9 Overovanie priepustnosti pipeline

```
cycle(pipeline[i+10000]) - cycle(pipeline[i]) =< t1
```

Príklad 5.10 Množstvo smerovaných paketov za určitý čas

```
cycle(forward[i+1000]) - cycle(forward[i]) =< t2
```

Autori hodnotia hlavné prínosy verifikácie založenej na formálnych tvrdeniach v možnosti podchytenia zložitejších podmienok a situácií. Výsledky často napomohli k overeniu funkčnosti a výraznému skvalitneniu návrhu, či už ide o skrátenie času potrebného na nachádzanie chýb, alebo počet odhalených nezrovnalostí v návrhu. Okrem toho, overovanie pomocou LOC prinieslo jasné a prehľadné výsledky výkonnostných analýz funkčných vlastností a iných kvantitatívnych dát. [3]

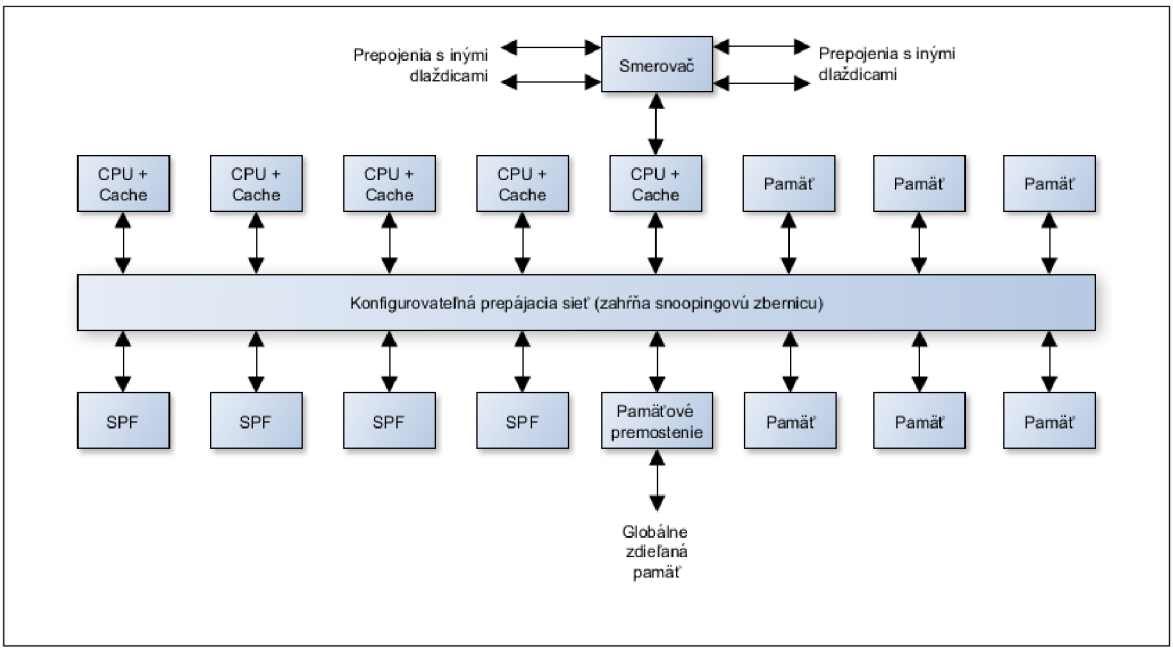
5.2 Prípadová štúdia verifikácie založanej na formálnych tvrdeniach pre multiprocesor SpaceCAKE

Prípadová štúdia [8] sa zaoberá aplikáciou verifikácie založenej na formálnych tvrdeniach na multimilión hradlovú architektúru SpaceCAKE so zdieľanou L2 cache pamäťou. Práca opisuje architektúru, problémy a výzvy, s ktorými sa autori stretli počas funkčnej verifikácie a zahŕňa popis využitých techník. Na účel verifikácie boli využité jazyk PSL a knižnica Open Verification Library.

SpaceCAKE (obrázok 7) je homogénna DSM architektúra (anglicky Distributed Shared Memory) zložená zo základných komponentov nazývaných dlaždice. Každú dlaždicu predstavuje multiprocesorový systém na čipe obsahujúci:

- viacero programoveteľných jadier s integrovanou L1 cache,
- zbernicovú sieť prepajajúcu jadrá,

- L2 cache,
- jednotku správy pamäte pre mimo-čipovú DDR pamäť.



Obrázok 7: Architektúra SpaceCAKE

Zbernicový snooping/sniffing je technika zaisťujúca koherenciu zdieľanej pamäte, pričom táto koherencia je zabezpečená naprieč dlaždicami, a preto sa s ňou nie je potrebné pri vývoji softvéru zaoberať. Dlaždice spolu komunikujú pomocou vysokorýchlostných smerovačov. V systéme sa nachádza viacero pamäťových bánk na zlepšenie konkurentnosti, čo ovplyvňuje celkovú priepustnosť. Okrem toho systém obsahuje jednotky slúžiace na špeciálne hardvérové funkcie.

Verifikácia protokolov dátových prenosov, ktorá daný protokol úplne popisuje v mieste rozhrania, zabezpečuje, že všetky pripojené jednotky sú automaticky kontrolované. Okrem plánovania sledov udalostí špecifikovaných protokolom sa overujú aj operácie čítania a zápisu, oneskorenia cyklov, vzťahy adries a dát.

Nezávislosť modulov návrhu síce poskytuje mnohé výhody, no prináša aj riziká. Tie je možné podchytiť pomocou formálnych tvrdení, ktoré kontrolujú príjem/vysielanie validných dát, vyjednávanie modelu správania a medzimodulovú komunikáciu.

V architektúre sa vyskytuje viacero typických štruktúr ktorými sú: fronta, vyrovnávacia pamäť, pamäť, arbiter, konečný automat a pipeline. Pri jednotlivých komponentoch sú uvedené ich hlavné overované charakteristiky:

- Fronta by nemala prijímať dáta, ak je plná, nemala by odmietať dáta, ak plná nie je a prepis dát by nemal spôsobovať pretečenie. Vo všeobecnosti by sa fronta mala správať štandardne, pričom pokrytie zisťuje pod/predimenzovanosť jej samotnej.
- Vyrovnávacia pamäť (buffer) nesmie naraz zapisovať a čítať z jedného miesta. Šírka zapisovaných dát je konzistentná so šírkou vkladaných dát, prepis dát pred ich čítaním je zakázaný a bufferové indexovanie čítania a zápisu musí byť konzistentné. Pokrytie odhaľuje prevažne štatistické informácie o vyťaženosti jednotlivých častí bufferu.

- Pamäte nesmú nikdy zapísať neplatné alebo neznáme hodnoty a po resete musia mať špecifikovanú hodnotu. Pri kontrole ich pokrytia sa zisťuje ich vyťaženosť a schopnosť zapisovať na všetky pamätové miesta.
- Pri arbiteroch sa kontroluje viacero ich nastaviteľných schém správania, pridelenie zdrojov najvyššej priorite, garantovanie pridelenia do určitého času a ich vyťaženosť.
- Konečné automaty potrebujú verifikáciu všetkých prechodov, kontrolu proti uviaznutiu v určitom stave/slučke stavov alebo proti uviaznutiu. Pokrývacie formálne tvrdenia poskytujú štatistické informácie o prechodoch a jednotlivých stavoch.
- Pipeline musí zabezpečiť spracovanie každej žiadosti, zamedziť výskytu poškodených dát alebo uviaznutiu. Pri pipeline sa pokrýva počet zahltení, počet a typ požiadaviek.

Vzhľadom na vysokú konfigurovateľnosť celého návrhu bol pri tvorbe testovacej sady kladený dôraz na znovupoužiteľnosť. Na vykonávanie požiadaviek čítania a zápisu bol využitý zbernicový funkčný model (anglicky Bus Functional Model, skrátene BFM), ktorý samotné inštrukcie nevykonáva, ale emuluje dianie na zbernici. Pri vykonávaní testu boli výsledky operácii zapisované okrem štandardných umiestnení aj do špeciálnej kontrolnej pamäte slúžiacej ako referenčná pamäť. Pri následnej práci s pamäťou bola kontrolovaná správnosť dát porovnávaním s hodnotami v referenčnej pamäti. Testovanie zahŕňalo viaceré testované konfigurácie kvôli zabezpečeniu dostatočnej verifikácie vzájomných prepojení jednotlivých prvkov.

Pomocou vyššie popísaných postupov sa autorom podarilo odhaliť viacero chýb v návrhoch modulov: nedostupný kód, nepovolené výstupné hodnoty alebo chyby vo vyhľadávaní IP adries. Autori dosiahli viacero ďalších významných výsledkov. Podarilo sa im znížiť čas „debugovania“ o 50%, dosiahli 95%-né pokrytie DSM funkcionality a dynamickou verifikáciou overili 62% formálnych tvrdení. [8]

5.3 Funkčná verifikácia procesorového chipsetu IBM System z10

Chipset IBM System z10 reprezentoval značnú zmenu návrhu oproti svojmu predchodcovi (IBM System z9). Potreba kompletného funkčného overenia nového chipsetu pred jeho uverejnením či výrobou priniesla nové výzvy v oblasti verifikácie.

Chipset IBM System z10 pozostáva zo štvorjadrového centrálneho procesoru z10 a z10 symetrického multiprocesoru. Hlavnými zmenami oproti predošlej verzii je nový návrh celého procesorového jadra, ktorého cieľom je poskytnúť výkonnú a úspornú implementáciu IBM architektúry *z*. Mikroprocesorová pipeline bola upravená kvôli podpore až dvojnásobnej operačnej frekvencie. Rovnako so sebou nový chipset priniesol aj vylepšenie predikcie vetiev, optimalizáciu cache hierarchie a hardvérovú implementáciu desiatkovej aritmetiky desatinných čísiel [7].

V článku [7] autori rozdeľujú samotný proces verifikácie produktu do troch fáz:

- podsystém procesorovej cache - pokrýva funkcie mimo jadra procesoru,
- verifikácia jedného procesorového jadra, zdieľanej cache druhej úrovne a ko-procesorovej jednotky,
- integrácia z10 chipsetu do úrovne systému.

Cache podsystém procesoru pozostáva z jednotky pamäťového kontroleru, GX jednotky rozhrania na vstupno/výstupný podsystém, PX jednotky zabezpečujúcej prepojenie medzi klientmi centrálneho procesoru a jednotky systémového kontroleru. V procese verifikácie na úrovni jednej či viacerých spomínaných jednotiek zohrávala úlohu simulácia založená na vyhodnocovaní každého cyklu (anglicky cycle-based). Naopak, pri overovaní blokov kritických pre návrh bola využitá formálna verifikácia.

Pre modely využívané pri verifikácii cache subsystému boli vytvorené komponentové balíčky, z ktorých sa vyskladal celkový model používaný na overovanie danej skupiny vlastností. Najvyššia úroveň predstavujúca jednu tzv. PU knihu (anglicky Processor Unit) obsahovala multichip balíček, pamäťový blok a vstupno/výstupný blok. Multichip balíček pozostával z dvoch systémových kontrolerov a piatich čipov centrálnych procesorov. Pomocou uvedeného balíčkového systému boli vytvorené modely s rôznymi parametrami, teda rôznym početom PU kníh a iných vnútorných komponentov.

V reálnom systéme, na rozdiel od simulácie, má vplyv na rýchlosť prenosu okrem dĺžky vodičov aj rýchlosť zbernice, a preto môžu vznikať špecifické situácie, ktoré nie je možné simulovať bez dodatočných opatrení. To pre autorov znamenalo potrebu vytvorenia schránok okolo samotných komponentov, v ktorých sa oneskorenia dali programovo nastavovať. Pomocou vytvorených modelov bolo možné otestovať aj väčšie štruktúry, čo autori článku [7] považujú za mimoriadne prínosné, pretože dané simulácie poskytli informácie o výkonnosti jednotlivých konfigurácií, ako aj umožnili zistiť ich výkonnostné hranice a obmedzenia.

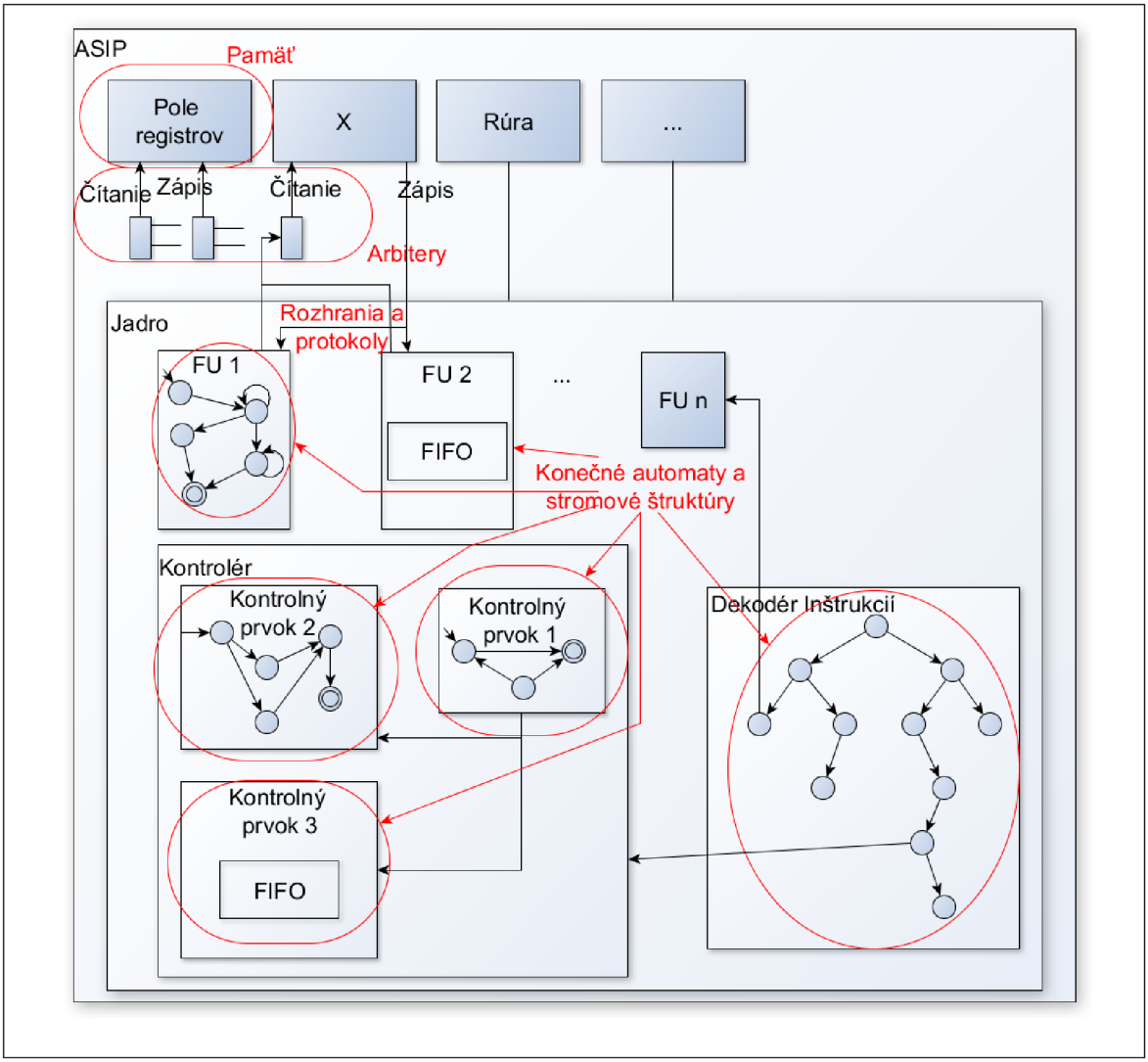
Poslednou časťou verifikačného procesu pri verifikovaní cache subsystému bolo aplikovanie formálnej verifikácie, ktorá pomohla odhaliť viaceré návrhové chyby.

Podobne ako pri predchádzajúcej fáze, aj v druhej etape bol aplikovaný hierarchický prístup. Postupným prechádzaním od najmenších a najjednoduchších blokov ku zložitejším bol vytvorený model celého jadra s vlastnou cache a zdieľanou koprocesorovou jednotkou.

Jednou z noviniek, s ktorou prišiel chipset od IBM bola jednotka na privolanie inštrukcie (anglicky Instruction Fetch Unit, skrátene IFU) a s ňou spojená predikcia vetiev. Hlavným stavebným prvkom overovania boli uzly, ktoré reprezentovali jednotlivé inštrukcie a udalosti, ktoré spôsobovali zmenu stavu IFU. Pomocou funkčnej verifikácie sa podarilo ďalej nájsť chyby v FPU (anglicky Floating Point Unit), overiť koherenciu pamäte a overiť správnosť návrhu architektúry.

Kapitola 6

Návrh formálnych tvrdení



Obrázok 8: Príklad architektúry

V kapitole návrh využívam schému 8, ktorá bola vytvorená ako zjednodušený model procesoru typu ASIP zachytávajúci časti podstatné pre verifikáciu. Na tejto schéme budú

postupne prezentované jednotlivé prvky, ktoré boli po konzultácii s vedúcou práce a s konzultantom vybrané ako najvhodnejšie pre verifikáciu založenú na formálnych tvrdeniach.

6.1 Rozhrania a protokoly

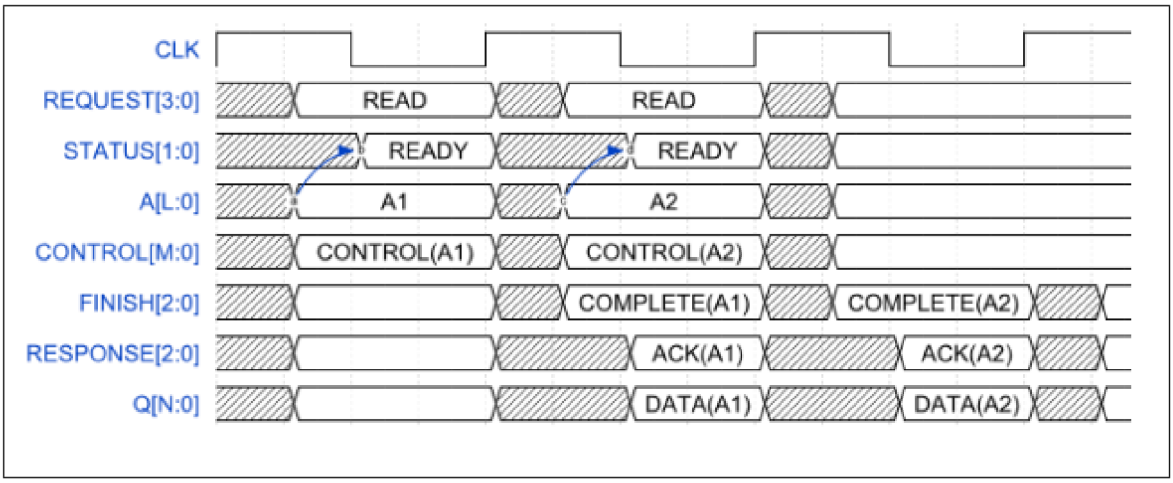
V návrhu je potrebné počítať aj s kontrolou rozhraní a protokolov využívaných pri vzájomnej komunikácii komponentov celého systému. Či už je, alebo nie je arbiter využívaný na správu zdieľaných prostriedkov, je potrebné, aby návrh zahŕňal aj kontrolu využívaných protokolov, teda správnosť sledu a formy signálov.

Procesor Codix RISC využíva zbernicu Codasip Local Bus (CLB) vytvorenú firmou Codasip. CLB je vysoko-výkonná synchrónna zbernica určená primárne na komunikáciu medzi Codix jadrami a perifériami ako pamäť, cache a periférie s vysokou priepustnosťou. V súčastnosti sa jedná o tzv. single-master systém, kedy iba jeden komponent vystupuje ako riadiaca jednotka (master), pričom sa v súčasnosti pracuje na podpore multi-master rozšírenia. Následne je popísaná krátka špecifikácia zbernice pre lepšie pochopenie jednotlivých signálov a celkového návrhu formálnych tvrdení v ďalších častiach práce.

Signály a ich popis:

- `REQUEST` - vstupný signál reprezentujúci požiadavku, jeho povolené hodnoty sú: None, Read, Write, Invalidate, Invalidate_All, Flush, Flush_All,
- `FINISH` - vstupný signál reprezentujúci stav ukončenia operácie, jeho povolené hodnoty sú: Complete, Continue,
- `STATUS` - výstupný signál reprezentujúci stav zariadenia, jeho povolené hodnoty sú: Busy, Ready, Error,
- `RESPONSE` - výstupný signál reprezentujúci odpoveď, jeho povolené hodnoty sú: Idle, Ack, Wait, Error, Unaligned, OOR,
- `A` - vstupný signál reprezentujúci adresu,
- `SI` - vstupný signál reprezentujúci index pod-bloku v rámci bloku dát,
- `SC` - vstupný signál reprezentujúci počet pod-blokov,
- `D` - vstupný signál reprezentujúci zapisované dáta,
- `Q` - výstupný signál reprezentujúci čítané dáta.

Obrázok 9 zobrazuje časový priebeh signálov pri požiadavke na čítanie dát. Jednotlivé signály sú výhodnocované pri nábežnej hrane hodinového signálu. Pri požiadavke na čítanie dát dôjde pri splnení dodatočných podmienok (zariadenie je pripravené, je nastavená platná adresa) k čítaniu dát na zariadení. Ak sa nevyskytnú problémy či chyby, v ďalšom hodinovom takte je na výstupe zariadenia privedená hodnota prečítaných dát, je signalizované potvrdenie ukončenej operácie a zmena stavu na stav reflektujúci pripravenosť zariadenia prijímať ďalšie požiadavky. Zároveň je možné zadať ďalšiu požiadavku.



Obrázok 9: Príklad komunikácie na zbernici - požiadavka na čítanie dát [16]

Pri verifikovaní zbernice pomocu formálnych tvrdení je potrebné sa zamerať na nasledujúce body:

- na zbernici sa môžu vyskytovať len platné dáta, konkrétnejšie, signály nesmú obsahovať X-ové hodnoty počas validných operácií,
- singály sa môžu vyskytovať iba v platnýh stavoch, resp. nesmú nadobúdať rezervované hodnoty počas validných operácií,
- hodnota adresy musí byť v platnom rozsahu, resp. nenastane prístup mimo povolený rozsah,
- na zbernici sa nevyskytnú požiadavky na nezarovnaný prístup do pamäte,
- v prípade neplatnej požiadavky (out-of-range, nezarovnaný prístup) bude hodnota signálu response správne indikovať chybnú požiadavku,
- ak na zbernici nie je registrovaná žiadna požiadavka, odpoveď pripojeného zariadenia reflektuje jeho nečinnosť,
- pri každej validnej požiadavke nastane validná odpoveď,
- každá požiadavka skončí buď potvrdením ukončenia spracovania požiadavky, alebo chybovým kódom,
- pri požiadavke na zaneprázdnené zariadenie nie je táto požiadavka spracovávaná a vykonáva sa blokovanie.

6.2 Pamäte

Dnes takmer každý moderný procesor potrebuje k svojej činnosti pamäť, a preto je potrebné v návrhu zahrnúť jej overovanie. Jednou zo situácií, ktoré môžu nastať, a ktoré by mali byť verifikované, sú neplatné a neznáme adresy. Pri ich vyhodnocovaní môžu nastať rôzne chyby ako sú neplatné dáta, získanie dát, ku ktorým by procesor nemal mať prístup, alebo neschopnosť zariadenia vyhodnotiť adresy.

Procesor Codix RISC má spoločnú pamäť pre dáta aj inštrukcie a na prístup do nej je využívaná 32-bitová adresa. Táto adresa odkazuje na blok dát skladajúci sa zo stanoveného počtu pod-blokov. V procesore využívanom v tejto práci predstavuje blok skupinu štyroch bajtov. Na prístup do týchto bajtov v rámci jedného bloku sa využívajú signály: Subblock index (`SI`) určujúci počiatočný čítaný či zapisovaný pod-blok v danom bloku a Subblock count (`SC`) predstavujúci počet pod-blokov v danom bloku, s ktorými sa bude ďalej pracovať. Pomocou nastavení týchto signálov je možné pristupovať a pracovať s dátami konkrétneho bajtu. Procesor Codix RISC má nastaviteľnú možnosť pre nezarovnaný prístup k pamäti, no táto možnosť je rovnako ako v procesore využívanom v tejto práci zakázaná. Z tohoto dôvodu by mal byť nezarovnaný prístup k pamäti kontrolovaný, pričom overovanie by malo zahŕňať kontrolu adresy, ako aj signálov `SI`, `SC` a ich vzájomného vzťahu.

Pamäť je s jadrom prepojená pomocou vyššie uvedenej zbernice CLB. Samotná pamäť nepodporuje všetky príkazy zbernice, a preto je potrebné zabezpečiť, že na požiadavky, ktoré pamäť nepodporuje bude odpovedané validnou odpoveďou, či už sa jedná o chybový kód alebo inú signalizáciu. Ďalšou úlohou verifikácie je zabezpečiť relevantnú odpoveď na validné požiadavky, to znamená, že na požiadavku na čítanie dát bude odpoveďou správna chybová hláška opisujúca príčinu chyby (napr. nezarovnaný prístup či adresa mimo povoleného rozsahu), požadované dáta, alebo odpoveď informujúca o spracovávaní požiadavky (požiadavka zaznamenaná, čakajte, zariadenie je zaneprázdnené, zariadenie je nečinné).

Kvôli spoločnej pamäti pre dáta a program je pamäť k procesoru pripojená dvomi rozhraniami. Prvé z nich, inštrukčné, má prístup iba na čítanie, tzv. read-only a druhé, dátové, má prístup na čítanie aj zápis tzv. read-write. Navrhované riešenie z dôvodu znovupoužiteľnosti predpokladá rozdelenie verifikácie pomocou formálnych tvrdení na časť overujúcu vlastnosti spojené s operáciou čítania a na časť overujúcu vlastnosti spojené so zápisom, prípadne vytvorenie časti overujúcej prácu s pamäťou bez ohľadu na to, či ide o čítanie alebo zápis.

6.3 Registre

Registre sú podstatnou a nevyhnutnou súčasťou procesorov a slúžia najmä ako veľmi rýchla pamäť, s ktorou procesor priamo pracuje. Najčastejšie sa využívajú na ukladanie výsledkov, načítavanie operandov, načítavanie dát z pomalších pamätí a ich opätovné ukladanie.

Podobne ako pri pamäti aj prácu s registrami je vhodné rozdeliť do viacerých častí podľa oprávnení na operácie, ktoré môžu byť s danými registrami vykonávané. Takéto rozdelenie teda vytvorí minimálne dve skupiny, konkrétne formálne tvrdenia overujúce operácie čítania a formálne tvrdenia overujúce zápis. Návrh predpokladá kontrolu:

- platných dát, tj. kontrola x-ových hodnôt,
- kontrola správneho adresovania, resp. kontrola prístupu mimo povolený rozsah adries,
- ochrana dát pri práci s nimi, tj. aspoň upozorniť na simultánne čitanie a zápis vykonávaný v jednom registri,
- upozornenie na zbytočné operácie čítania, teda z viacerých portov na čítanie sa číta hodnota z rovnakej adresy v rovnakom čase.

6.4 Ostatné komponenty systému

Arbiter ako prvok, ktorý má na starosti riadenie a prideľovanie zdieľaných zdrojov viacerým komponentom, je ďalším prvkom vybraným pre tento spôsob verifikácie. Rôzne modely ich správania sú popísané v predchádzajúcej kapitole. Hlavnými overovanými vlastnosťami arbitrov musia byť:

- neschopnosť poskytnúť zariadeniu prístup ku zdieľanému zdroju v rozumnom čase, čo často značí možnosť uviaznutia,
- ochrana pred chybami v modeloch správania ako sú neuprednostnenie požiadavky s vyššou prioritou, povolenie prístupu viacerým vzájomne výlučným prvkom, povolenie prístupu aj napriek vyčerpaniu kreditov apod.

Konečné automaty a stromové štruktúry predstavujú riadiacu logiku kontrolérov, funkčných jednotiek a iných prvkov. Kontrolované budú nepovolené sekvencie prechodov, zacyklenie, neplánované prechody do koncových stavov, nesprávne alebo nevykonané prechody. Stromové štruktúry môžu predstavovať riadiacu logiku ako napríklad v prvku dekodér na obrázku 8. Návrh počíta s kontrolou neplatných vstupných sekvencií, či nesprávnych vyhodnotení, kedy sa napríklad indikuje chybná sekvencia aj napriek jej správnemu vyhodnoteniu. Príkladom prvku riadiacej logiky je dekodér inštrukcií. Prvou overovanou vlastnosťou by malo byť obmedzenie x-ových hodnôt vstupných a výstupných signálov. Následne by bolo vhodné zaoberať sa výskytom neznámych inštrukcií.

Kapitola 7

Implementácia

Táto kapitola sa venuje popisu implementácie návrhu uvedeného v predchádzajúcej kapitole a postupne čitateľovi poskytuje náhľad na riešenia jeho jednotlivých častí. V prvom rade popisuje prostriedky použité v práci, tvorbu verifikačného prostredia a rozdelenie práce do funkčných celkov. Ďalej popisuje jednotlivé časti, k nim príslušné formálne tvrdenia z ktorých časť bola vytvorená mnou a časť bola na základe môjho návrhu a dodatočnej konzultácii implementovaná firmou Codasip. V kapitole sa následne venujem vytvoreniu zrozumiteľného postupu pomocou ktorého by bolo možné uľahčiť ladenie návrhu samotného procesoru pomocou formálnych tvrdení a zjednodušiť používanie vytvoreného verifikačného prostredia pre daný procesor. Kapitolu uzatvárajú analýzy pokrytia formálnych tvrdení a celkové zhodnotenie výsledkov.

7.1 Tvorba verifikačného prostredia

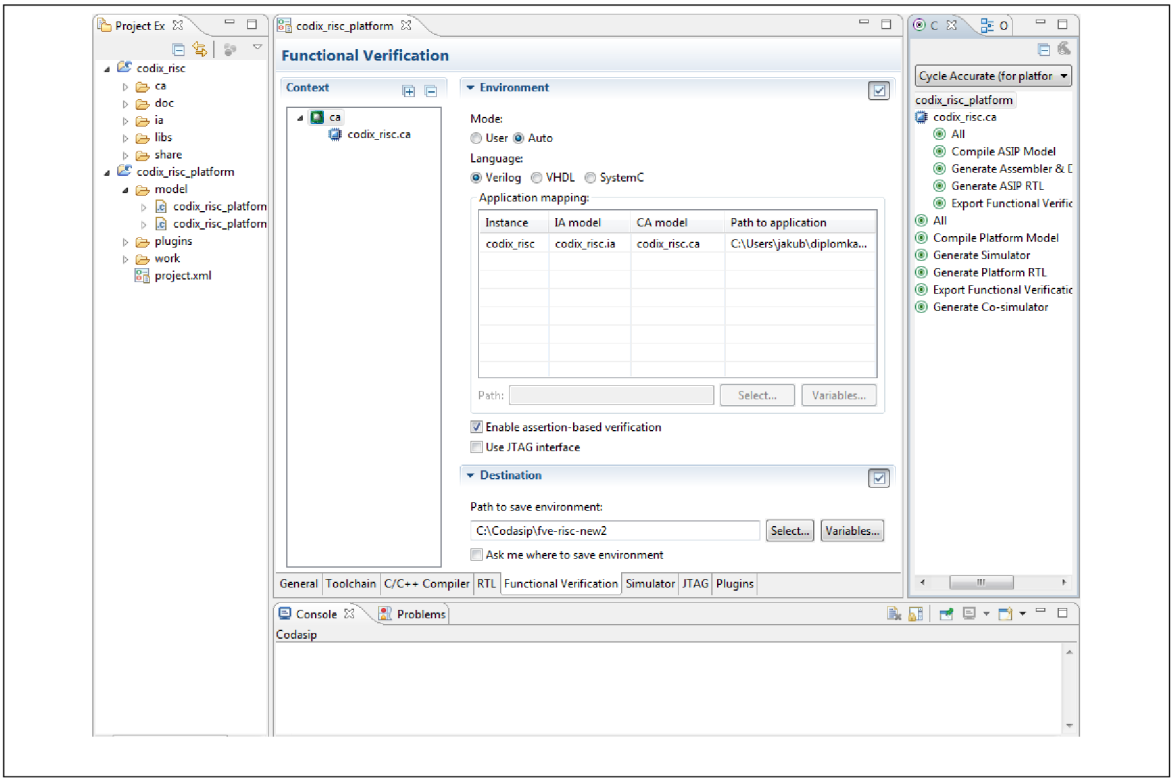
Na získanie modelu zvoleného procesoru Codix RISC je potrebná registrácia na portáli firmy Codasip[16] a následné zažiadanie o prístupové práva na model procesoru, jeho platformu a ďalší potrebný softvér. Po nastavení komponentov potrebných pre beh Codasip frameworku (licenčný server a middleware) je možné spustiť samotný framework a naimportovať Codix platformu a procesor Codix RISC získaný zo spomínaného webu. Pre vygenerovanie HDL popisu procesoru potom stačí nastaviť parametre generovania ako sú jazyk, v ktorom bude výsledný model popísaný, mód generovania a mapovanie aplikácie (obrázok 10), ktoré následne na danom procesore pobeží.

Model procesoru využívaný v práci je popísaný v jazyku Verilog a zvolili sme automatické generovanie verifikačného prostredia pri ktorom je následne nutné dodať cestu k aplikácii, ktorá beží na procesore. Vstupné aplikácie musia byť spustiteľné binárne súbory alebo ich zkomprimované „zip“ verzie. Tieto aplikácie boli vybrané v spolupráci s vedúcou práce ako vhodní kandidáti spomedzi dostúpných apliikácií existujúcich pre daný procesor.

Generovanie HDL popisu je možné spustiť pre procesor samotný alebo pre celú platformu obsahujúcu Codix RISC procesor a pamäť. Návrh verifikácie sa okrem procesoru zaoberá aj pamäťou pretože je jeho neodmysliteľnou súčasťou, ktorú procesor aplikovaný do praxe pre svoju činnosť potrebuje. Preto bolo zvolené verifikačné prostredie pre celú platformu namiesto generovania prostredia pre procesor samotný.

Tvorbe verifikačného prostredeia predchádza kompilácia modelu procesoru, generovanie assembleru a disassembleru, vytvorenie HDL popisu pre ASIP, kompilácia celej platformy, tvorba simulátora a HDL popisu celej platformy a následne export vytvoreného verifikač-

ného prostredia (obrázok 10).



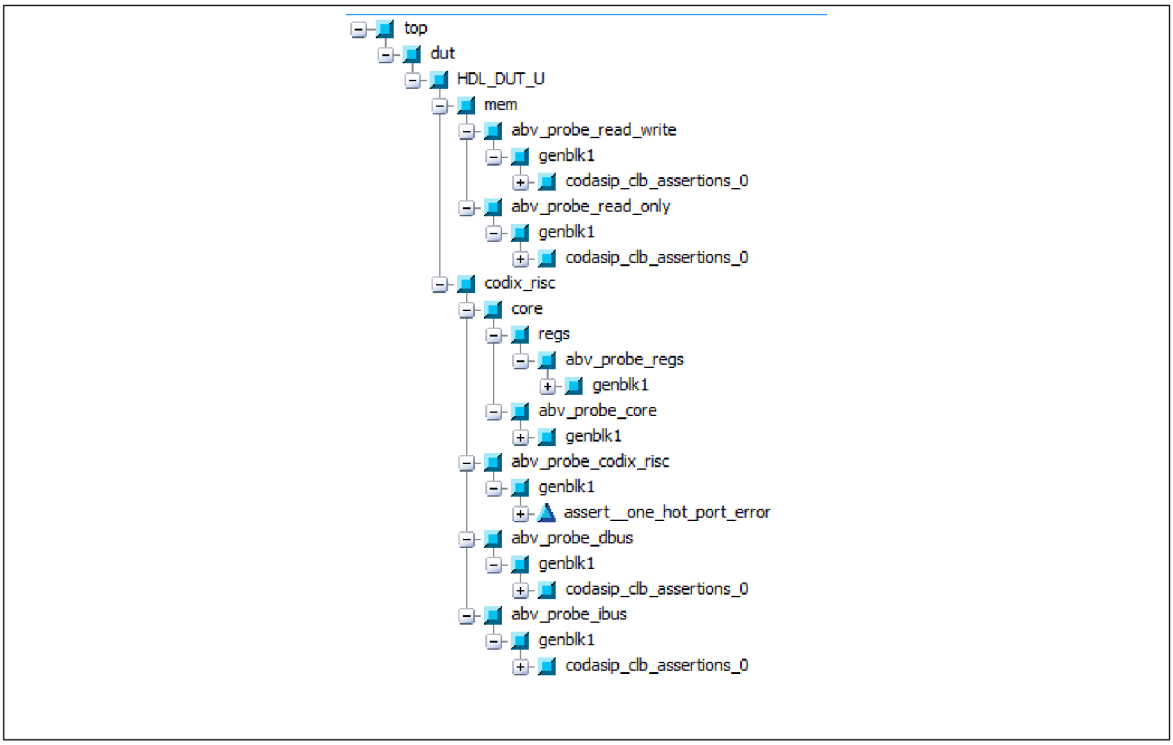
Obrázok 10: Codasip Framework [16]

Súborová štruktúra vygenerovaného verifikačného prostredia je logicky rozdelená na popis platformy v zvolenom jazyku a funkčné verifikačné prostredie obsahujúce skripty na spustenie simulácie, konfiguračné súbory, mapované aplikácie, tzv. golden model a agentov. Každý z agentov reprezentuje logický celok verifikácie s definovaným rozhraním určeným pre verifikáciu. Formálne tvrdenia je možné teda priamo dopisovať do jednotlivých agentov, presnejšie do zdrojových súborov, ktoré sú na tieto účel vytvorené.

V práci je využívaný simulátor QuestaSim [19] od spoločnosti Menthor Graphics. Po konzultácii s vedúcou práce a verifikačným tímom spoločnosti Codasip sme sa rozhodli nepoužiť nástroj Modelsim ale jeho modernejšieho nástupcu, ktorý zo spomínaného Modelsimu vychádza. Hlavnou výhodou použitia nástroja QuestaSim je rozšírenie možností debugovania hradlových polí a systémoch na čipe (anglicky System on Chip, skrátene SoC) a vylepšená podpora jazyku SystemVerilog a OVM/UVM metodiky. Pomocou nástroja QuestaSim je možné spustiť simuláciu vygenerovaného verifikačného prostredia. Simulácia procesoru je spustená s aplikáciami, ktoré boli namapované na daný procesor.

V práci som spolupracoval so zamestnancami firmy Codasip [16] na tvorbe formálnych tvrdení pre spomínaný procesor. Vo firme bol navrhnutý systém generovania formálnych tvrdení zo šablón. Tieto šablóny sú vstupmi pre tzv. sondy (anglicky probes), využívajúce už spomínanú konštrukciu `generate` na generovanie formálnych tvrdení z týchto šablón. Šablóny obsahujú všeobecné formálne tvrdenia rozdelené do viacerých skupín ako sú: formálne tvrdenia pre CLB zbernicu, formálne tvrdenia overujúce pamäť a formálne tvrdenia overujúce prácu s registrami. Sondy na základe nastavených parametrov, ako napríklad povolenie verifikácie založenej na formálnych tvrdeniach, generujú formálne tvrdenia a každá sonda má priradenú určitú skupinu šablón, s ktorými pracuje. Štruktúru jednotlivých sond

a formálnych tvrdení, ktoré obsahujú je možné vidieť na obrázku 11.



Obrázok 11: Sondy obsahujúce vygenerované formálne tvrdenia

7.2 Fronta

Fronta je prvok bežne sa vyskytujúci pri spracovávaní a uchovávaní informácií, kde často slúži ako vyrovnávacia pamäť na vstupe a výstupe rôznych komponentov. Pri práci na tvorbe formálnych tvrdení pre frontu som vďaka firme Codasip získal dočasne prístup aj k produktu OneSpin 360 firmy OneSpin Solutions, pomocou ktorého som na samostatnom modeli fronty vytvoril a testoval formálne tvrdenia.

Pri verifikácii fronty bol kladený dôraz na znovupoužiteľnosť formálnych tvrdení kvôli jej častému výskytu nielen v návrhu procesoru Codix RISC, ale aj pri rôznych externých komponentoch návrhu. Formálne tvrdenia pracujúce s frontou predpokladajú existenciu signálov `empty` a `full`, ktoré signalizujú prázdnu, resp. plnú frontu. Tieto signály nie sú vyžadované pre samotnú činnosť fronty, no často sa v nej nachádzajú ako kontrolný mechanizmus podtečenia/pretečenia. Pri čítaní a zápise je možné ich do návrhu doplniť (napr. odvodiť ich hodnotu z hodnoty ukazateľov čítania/zápisu). Niektoré zo základných formálnych tvrdení (znázornené na príkladoch 7.1 a 7.2) overujúcich frontu sú také, ktoré overujú naplnenosť fronty, pretečenie resp. podtečenie a prípadné súčasné vykonávanie zápisu resp. čítania pri stave plnej/prázdnej fronty, správnosť čítaných či zapisovaných dát a pod.

Vyššie uvedené formálne tvrdenia pokrývajú len základné vlastnosti návrhu, no na komplexnejšie pokrytie je potrebné overovať aj situácie, ktorých výskyt sa v návrhu nepredpokladá a správanie systému pri štarte/resete. V príklade 7.3 je znázornené formálne tvrdenie kontrolujúce súčasný výskyt signálu `empty` a `full` signalizujúci prázdnu resp. plnú frontu a vynulovanie potrebných signálov pri resete.

Príklad 7.1 Základné formálne tvrdenia pre frontu 1

```
// V prípade že ide o frontu, ktorá je chránená pred pretečením
// Fronta je plná, a preto nesmie byť umožnený zápis
property no_write_on_full;
    @(posedge clk) disable iff (!fifo.reset_n)
        fifo.full |=> !fifo.wr_en;
endproperty
// Fronta je prázdna, a preto nesmie byť umožnené čítanie
property no_read_on_empty;
    @(posedge clk) disable iff (!fifo.reset_n)
        fifo.empty |=> !fifo.rd_en;
endproperty
```

Príklad 7.2 Základné formálne tvrdenia pre frontu 2

```
// Kontrola výstupných dát pri čítaní
property correct_data_read;
    @(posedge clk) disable iff (!fifo.reset_n)
        fifo.rd_en |-> fifo.data_out == mem(fifo.rd_addr);
endproperty
// Kontrola zapisovaných dát
property correct_data_write;
    @(posedge clk) disable iff (!fifo.reset_n)
        fifo.wr_en |-> fifo.data_in == mem(fifo.wr_addr);
endproperty
```

Príklad 7.3 Kontrola plnej/prázdnej fronty a hodnoty signálov po resete

```
// Fronta nikdy nesmie byť plná a zároveň prázdna
property fifo_full_and_empty;
@(posedge clk) disable iff (!fifo.reset_n)
    !(fifo.full && fifo.empty);
endproperty

// Fronta vynuluje a nastaví potrebné signály do počiatočného stavu
property on_reset;
@(posedge clk)
    !fifo.reset_n |=> fifo.empty && !fifo.full &&
    (wr_addr == zero_pointer) && (rd_addr == zero_pointer);
endproperty
```

Posledná skupina (príklady 7.4 a 7.5) overuje komplexnejšie správanie sledovaním sekvencií v čase. Pomocou týchto formálnych tvrdení je overované nesprávne nastavovanie signálov `full` a `empty`, naplňovanie a vyprázdňovanie fronty, prechod medzi týmito stavmi a ďalšie vlastnosti, ktorých overovanie vyžaduje kontrolu hodnôt v časových úsekoch pomocou sekvencií.

Príklad 7.4 Overovanie dosiahnutia plnej fronty

```
// Ak sa fronta zaplnila, v minulom takte musel nastať zápis a nesmelo
// nastať čítanie
property fifo_full_previous_write;
@(posedge clk) disable iff (!fifo.reset_n)
    $rose(fifo.full) |-> $past(fifo.wr_en && !fifo.rd_en);
endproperty
```

Príklad 7.5 Overovanie prázdnej fronty a súvisiacich vlastností

```
// Ak sa fronta vyprázdnila, v minulom takte nesmel nastať zápis a
// muselo nastať čítanie
property fifo_empty_previous_read;
@(posedge clk) disable iff (!fifo.reset_n)
    $rose(fifo.empty) |-> $past(!fifo.wr_en) && $past(fifo.rd_en);
endproperty
// Ak je fronta prázdna, môže sa naplniť najskôr po FIFO_DEPTH zápisoch
property fifo_from_empty_to_full;
@(posedge clk) disable iff (!fifo.reset_n)
    fifo.empty |=> !(fifo.full) throughout (fifo.wr_en && !fifo.rd_en)
    [->FIFO_DEPTH];
endproperty
// Stabilita signálov full a epmty
property fifo_stable_empty_full;
    @(posedge CLK) disable iff (!fifo.reset_n)
    fifo.rd_en == fifo.wr_en |=> $stable(fifo.full) && $stable(fifo.empty);
endproperty
```

7.3 Pamäť

Pamäť procesoru Codix RISC implementuje dve rozhrania typu CLB (Codasip Local Bus), kde pamäť vystupuje ako prijímacia strana komunikácie (anglicky slave). Prvé rozhranie poskytuje prístup iba pre čítanie (anglicky read-only) a je napojené na inštrukčnú zbernicu `ibus`.

Druhé rozhranie pamäte poskytuje prístup pre čítanie aj zápis (anglicky read-write) a je pripojené na dátovú zbernicu procesoru `dbus`. V pamäti procesoru Codix RISC je možné nastaviť nezarovnaný prístup do pamäte, využíva sa endianita typu „little endian“ a latencia operácií čítania aj zápisu je jeden hodinový takt.

Implementácia formálnych tvrdení pre pamäť spočívala vo viacerých krokoch. Keďže v aktuálnej implementácii procesoru je nezarovnaný prístup do pamäte zakázaný, prvým krokom bolo vytvorenie základných formálnych tvrdení overujúcich zápis mimo prístupný blok dát 7.6 a nezarovnaný prístup k pamäti 7.7.

Príklad 7.6 Zápis a čítanie mimo prístupného bloku dát

```
// Počet buniek a index prvej požadovanej nesmú presiahnuť veľkosť bloku
property read_outside_byte;
    @(posedge CLK) disable iff (!reset)
        rd_en |-> ($unsigned(read_subblock_count) +
        $unsigned(read_subblock_index)) > SUBBLOCK_COUNT;
endproperty
// Počet buniek a index prvej zapisovanej nesmú presiahnuť veľkosť bloku
property write_outside_byte;
    @(posedge CLK) disable iff (!reset)
        wr_en |-> ($unsigned(write_subblock_count) +
        $unsigned(write_subblock_index)) > SUBBLOCK_COUNT;
endproperty
```

Príklad 7.7 Nezarovnaný prístup k pamäti

```
// Nezarovnaný prístup k pamäti pri čítaní
property unaligned_read;
    @(posedge CLK) disable iff (!reset)
        rd_en |-> ($unsigned(rd_addr) % SUBBLOCK_COUNT)  != 0;
endproperty
// Nezarovnaný prístup k pamäti pri zápise
property unaligned_write;
    @(posedge CLK) disable iff (!reset)
        wr_en |-> ($unsigned(wr_addr) % SUBBLOCK_COUNT)  != 0;
endproperty
```

Zbernica pripojená k pamäti je univerzálnou zbernicou, ktorej možnosti prevyšujú potreby použitej pamäte, a preto je potrebné kontrolovať, či do pamäte neprichádzajú nesprávne požiadavky, ktoré s činnosťou pamäte nesúvisia, ktoré sú neplatné, alebo pre ktoré pamäť nemá implementované požadované správanie. Ukážky takýchto formálnych tvrdení je možné nájsť v príkladoch 7.8 a 7.9, na ktorých je vidieť premennú `SUPPORTED`, ktorú kvôli

znovupoužiteľnosti zaviedla firma Codasip. Táto premenná je vo vyššie spomínanej sonde vypočítaná podľa požiadaviek danej overovanej komponenty a aktuálnej hodnoty signálu `REQUEST`.

Príklad 7.8 Overovanie kompatibility požiadaviek 1

```
// Neznáma / rezervovaná požiadavka
property memory_reserved_request;
    @(posedge CLK)
        !$isunknown(REQUEST) |-> REQUEST != 3;
endproperty
// Obmedzenie známych požiadaviek
property memory_unsupported_request;
    @(posedge CLK)
        !$isunknown(REQUEST) |-> SUPPORTED || REQUEST == CP_RQ_NONE;
endproperty
```

Príklad 7.9 Overovanie kompatibility požiadaviek 2

```
// Neznámy / rezervovaný status
property memory_reserved_status;
    @(posedge CLK)
        !$isunknown(STATUS) |-> STATUS != 2;
endproperty
// Výskyt chybového kódu v statuse
property memory_error_status;
    @(posedge CLK)
        !$isunknown(STATUS) |-> (STATUS != CP_ST_ERROR);
endproperty
```

Podľa druhu operácií, ktoré formálne tvrdenia pre pamäť overujú, ich ďalej delíme na : formálne tvrdenia pre čítanie, formálne tvrdenia pre zápis a formálne tvrdenia všeobecné a spoločné pre obe predošlé kategórie zároveň. Príklady zobrazené v 7.10 verifikujú adresovanie „out of range“, pri ktorom je vyžadovaná práca s neprístupným či neexistujúcim miestom v pamäti. Šírka adresy je v procesore 32b a veľkosť pamäte je pre procesor štandardne nastavená na 524Kb, teda rozsah adresy značne prevyšuje bežné využívanie pamäte. Následne je znázornené formálne tvrdenie overujúce dokončenie prebiehajúcej operácie pred požiadaním o ďalšiu, ktoré bolo doimplementované firmou Codasip.

Čítanie aj zápis predstavujú z pohľadu návrhu verifikačného systému dve oddelené časti, ktoré síce obsahujú spoločné vlastnosti zachytené v predošlej časti, no zároveň má každá svoje špecifická ako napríklad potenciálne rôznu latenciu týchto operácií, a preto boli tieto časti oddelené. Ďalším podnetom pre oddelenie celkov bola možnosť kombinovania typov prístupu do pamäte len na čítanie alebo na čítanie a zápis. Pre formálne tvrdenia spoločné pre operácie čítania aj zápisu bola vytvorená samostatná skupina formálnych tvrdení, zobrazená na príklade 7.11.

Pre operácie zápisu bola vytvorená samostatná skupina formálnych tvrdení, ktorá okrem správneho priebehu operácií verifikuje správnosť statusov, požiadaviek a odpovedí. Príklad 7.12 reprezentuje zástupcov z každej skupiny (formálne tvrdenia overujúce statusy, odpo-

Príklad 7.10 Formálne tvrdenia spoločné pre operácie čitania aj zápisu

```
// Ochrana proti použitiu neexistujúcej či neprístupnej pamäti
property memory_region_not_supported;
    @(posedge CLK)
        SUPPORTED && (STATUS == STATUS_READY) |-> (ADDR < MAX_ADDR);
endproperty
// Kontrola dokončenia jednej požiadavky pred začatím druhej
property memory_request_not_processed;
    @(posedge CLK)
       SUPPORTED && REQUEST == CP_RQ_WRITE_MEM || REQUEST == CP_RQ_READ_MEM
       && STATUS == CP_ST_READY |-> ##[0:1] REQUEST == CP_RQ_NONE &&
       STATUS != CP_ST_READY [*0:$] |=> STATUS == CP_ST_READY;
endproperty
```

Príklad 7.11 Formálne tvrdenia spoločné pre operácie čitania

```
// Indikácia čítania chybných dát
property memory_correct_data_read;
    @(posedge CLK)
        !$isunknown(IFRESPONSE) && (IFRESPONSE == RESPONSE_ACK) |->
        !$isunknown(Q);
endproperty
// Hodnota RESPONSE indikuje chybu UNALIDNED pri nezarovnanom prístupe
property memory_unaligned_response;
    @(posedge CLK)
        !$isunknown(IFRESPONSE) |-> (IFRESPONSE != RESPONSE_UNALIGNED);
endproperty
// Hodnota RESPONSE indikuje požadovanie nevalidnej pamäte
property memory_out_of_memory_response;
    @(posedge CLK)
        !$isunknown(IFRESPONSE) |-> (IFRESPONSE != RESPONSE_OOR);
endproperty
// Správnosť priebehu operáci čítania
property memory_read_ok;
    @(posedge CLK) disable iff (RST === CODASIP_RST_ACT_LEVEL)
       !$isunknown(REQUEST) && !$isunknown(SI) && !$isunknown(SC) &&
       SUPPORTED && (REQUEST==CP_RQ_READ_MEM) && (STATUS == CP_ST_READY) |->
       ##[READ_LATENCY_LOW:READ_LATENCY_HIGH] ((IFRESPONSE == CP_RC_ACK) ||
       (IFRESPONSE == CP_RC_ERROR) || (IFRESPONSE == CP_RC_UNALIGNED) ||
       (IFRESPONSE == CP_RC_OOR));
endproperty
```

vede, validitu dát apod.). Prvé formálne tvrdenie bolo vytvorené v rámci diplomovej práce a zvyšné príklady v danej skupine boli doimplementované firmou Codasip.

Príklad 7.12 Formálne tvrdenia spoločné pre operácie zápisu

```
// Ochrana proti zapisovaniu neznámych dát
property memory_correct_write_data;
    @(posedge CLK)
    (REQUEST == REQUEST_WRITE) && (STATUS == STATUS_READY) |->
        !$isunknown(DATA);
endproperty
// Operácia zápisu prebehne v˜poriadku a vráti validnú odpoveď
property memory_write_ok;
    @(posedge CLK) disable iff (RST === CODASIP_RST_ACT_LEVEL)
        !$isunknown(REQUEST) && !$isunknown(SI) && !$isunknown(SC)
        && SUPPORTED && (REQUEST == CP_RQ_WRITE_MEM) && (STATUS ==
        CP_ST_READY) ##(WRITE_LATENCY-1) 1 |->
        ((OFRESPONSE == CP_RC_ACK) || (OFRESPONSE == CP_RC_ERROR) ||
        (OFRESPONSE == CP_RC_UNALIGNED) || (OFRESPONSE == CP_RC_OOR));
endproperty
// Ak sa objaví nezarovnaná adresa, na OFRESPONSE sa privedie
// hodnota UNALIGNED
property memory_unaligned_not_set_on_ofrsp;
    @(posedge CLK) disable iff (RST === CODASIP_RST_ACT_LEVEL)
        !$isunknown(REQUEST) && !$isunknown(STATUS) && !$isunknown(A)
        && SUPPORTED && (REQUEST == CP_RQ_WRITE_MEM) && (STATUS ==
        CP_ST_READY) && (A % SBC != 0)
        ##(WRITE_LATENCY-1) 1 |-> (OFRESPONSE == CP_RC_UNALIGNED);
endproperty
// Ak sa objaví nepodporovaný REQUEST, na IFRESPONSE sa privedie
// hodnota IDLE
property memory_unsupp_ifresponse_idle;
    @(posedge CLK) disable iff (RST === CODASIP_RST_ACT_LEVEL)
        !$isunknown(REQUEST) && !$isunknown(STATUS) &&
        !SUPPORTED && (REQUEST != CP_RQ_NONE_MEM) && (STATUS ==
        CP_ST_READY) |-> ##[READ_LATENCY_LOW:READ_LATENCY_HIGH]
        (IFRESPONSE == CP_RC_IDLE);
endproperty
```

7.4 Zbernica Codasip Local Bus

Formálne tvrdenia pre zbernicu, tak ako pre pamäť, sú rozdelené do viacerých skupín, z ktorých niektoré boli vytvorené v rámci tejto práce a ostatné boli doimplementované na základe môjho návrhu firmou Codasip. Týmito skupinami sú:

- overovanie výskytu chybových kódov na portoch `STATUS` a `RESPONSE`,

- správna odpoveď `RESPONSE` na požiadavku `REQUEST` po jednom takte hodín od zadania tejto požiadavky,
- nastavovanie hodnoty `STATUS` pri odpovedi `WAIT`,
- kontrola požadovania iba povolených požiadaviek `REQUEST`,
- použitie rezervovaných hodnôt na rôznych portoch,
- overovanie hodnoty vstupných a výstupných dát `D` a `Q`,
- správnosť adresy `ADDR` a výberových signálov `SI`, `SC` pri správnej požiadavke `REQUEST`,
- spracovanie požiadavky `REQUEST` a poskytnutie relevantnej odpovede `RESPONSE` na validné požiadavky,
- ochrana pred nezarovnaným prístupom, pred prístupom do pamäte mimo povolený rozsah a pred prístupom mimo vyhradený blok dát.

Pri implementácii formálnych tvrdení na overenie správnej funkčnosti zbernice CLB som využil premennú `SUPPORTED`, ktorá bola pôvodne počítaná v sonde a predávaná ako vstupný signál generovaným formálnym tvrdeniam. V neskoršej implementácii sond spoločnosti Codasip sa daná premenná prestala nastavovať v sonde samotnej a aktuálne je počítaná v jednotlivých vzoroch formálnych tvrdení, kde plnia obdobnú úlohu. K tejto premennej bola pre potreby overovania zbernice doplnená ďalšia, `NOT_FORBIDDEN_REQUEST`, pomocou ktorej je kontrolovaná platnosť požiadavky na daný typ zbernice (viď príklad 7.13).

Príklad 7.13 Formálne tvrdenia overujúce zbernicu CLB - overovanie platnosti požiadavky

```
// Počítanie povolených požiadaviek na základe nastavenia zbernice
assign NOT_FORBIDDEN_REQUEST =
    ((FLAG == FLAG_RO) || (FLAG == FLAG_R)) ? (REQUEST == CP_RQ_READ) ||
    (REQUEST == CP_RQ_INVALIDATE) || (REQUEST == CP_RQ_INVALIDATE_ALL) :
    ((FLAG == FLAG_WO) || (FLAG == FLAG_W)) ? (REQUEST == CP_RQ_WRITE) ||
    (REQUEST == CP_RQ_FLUSH) || (REQUEST == CP_RQ_FLUSH_ALL) :
    (REQUEST == CP_RQ_READ) || (REQUEST == CP_RQ_INVALIDATE) ||
    (REQUEST == CP_RQ_INVALIDATE_ALL) || (REQUEST == CP_RQ_WRITE) ||
    (REQUEST == CP_RQ_FLUSH) || (REQUEST == CP_RQ_FLUSH_ALL);
// Kontrola platnosti operácie - ak požiadavka patrí medzi platné,
// výsledok je závislý od aktuálnych nastavení zbernice
property clb_supported_request;
    @(posedge CLK)
    !$isunknown(REQUEST) && (REQUEST == CP_RQ_READ || REQUEST ==
    CP_RQ_WRITE || REQUEST == CP_RQ_INVALIDATE || REQUEST ==
    CP_RQ_INVALIDATE_ALL || REQUEST == CP_RQ_FLUSH || REQUEST ==
    CP_RQ_FLUSH_ALL) |-> NOT_FORBIDDEN_REQUEST;
endproperty
```

Z vyššie uvedených skupín formálnych tvrdení sú tie vytvorené v rámci práce následne popísané na príkladoch 7.14 a 7.15. Tie znázorňujú formálne tvrdenia kontrolujúce získanie

odpovede pri zadaní požiadavky od pripojeného komponentu, pričom pre tento komponent je daná požiadavka validná. Druhé a tretie formálne tvrdenia overujú ochranu pred používaním rezervovaných hodnôt.

Príklad 7.14 Formálne tvrdenia overujúce zbernicu CLB

```
// Odpoveď na validnú požiadavku musí byť validná
property clb_correct_response;
    @(posedge CLK) disable iff (!RST)
       SUPPORTED && (STATUS == STATUS_READY) |=>
       ((RESPONSE == RESPONSE_ACK) || (RESPONSE == RESPONSE_ERROR) ||
       (RESPONSE == RESPONSE_UNALIGNED) || (RESPONSE == RESPONSE_OOR) ||
       (RESPONSE == RESPONSE_IDLE))
       or
       ((RESPONSE == RESPONSE_WAIT)[*1:$] ##1 ((RESPONSE == RESPONSE_ACK) ||
       (RESPONSE == RESPONSE_ERROR) || (RESPONSE == RESPONSE_UNALIGNED) ||
       (RESPONSE == RESPONSE_OOR) || (RESPONSE == RESPONSE_IDLE)));
endproperty
```

Príklad 7.15 Formálne tvrdenia overujúce zbernicu CLB

```
// Kontrola platných hodnôt na porte STATUS
property clb_reserved_1;
    @(posedge CLK)
        !$isunknown(STATUS) |-> (STATUS == STATUS_BUSY ||
        STATUS == STATUS_READY || STATUS == STATUS_ERROR);
endproperty
// Kontrola platných hodnôt na porte FINISH
property clb_reserved_2;
    @(posedge CLK)
        !$isunknown(FINISH) |-> (FINISH == FINISH_COMPLETE ||
        FINISH == FINISH_CONTINUE);
endproperty
```

Z ďalších skupín sú vhodnými príkladmi formálne tvrdenia implementované firmou Codasip. Vybrané formálne tvrdenia zobrazené v príklade 7.16 verifikujú nastavenie odpovede `IDLE`, ak má požiadavka hodnotu `NONE` a druhý z príkladov overuje nastavenie statusu na `BUSY` v prípade, že komponent na požiadavku odpovedal hodnotou `WAIT`.

Okrem samotného overovania nezarovnaného prístupu, ktoré bolo vytvorené v rámci práce, firma Codasip dodala aj formálne tvrdenie zabezpečujúce validnú odpoveď na takúto chybu. Konkrétne formálne tvrdenie je uvedené v príklade 7.17.

Príklad 7.16 Formálne tvrdenia overujúce zbernicu CLB

```
property clb_wrong_resp_on_none_request_ready_error;
    @(posedge CLK) disable iff (!RST)
        !$isunknown(REQUEST) && !$isunknown(STATUS) && (REQUEST ==
        CP_RQ_NONE) && (STATUS == CP_ST_READY || STATUS == CP_ST_ERROR) |=>
        (RESPONSE === CP_RC_IDLE);
endproperty
// Ak je odpoveď WAIT, status je nastavený na hodnotu BUSY
property clb_wait_and_busy;
    @(posedge CLK)
        !$isunknown(RESPONSE) && !$isunknown(STATUS) &&
        (RESPONSE == CP_RC_WAIT) |-> (STATUS == CP_ST_BUSY);
endproperty
```

Príklad 7.17 Formálne tvrdenia overujúce zbernicu CLB

```
// Pri nezarovnanom prístupe má odpoveď hodnotu \{tt CP_RC_UNALIGNED}
property clb_unaligned_address_unaligned;
    @(posedge CLK) disable iff (RST === CODASIP_RST_ACT_LEVEL)
        !$isunknown(REQUEST) && !$isunknown(STATUS) && !$isunknown(A) &&
        SUPPORTED && (STATUS == CP_ST_READY) && (REQUEST == CP_RQ_READ ||
        REQUEST == CP_RQ_WRITE || REQUEST == CP_RQ_INVALIDATE ||
        REQUEST == CP_RQ_FLUSH) && (A % SBC != 0) |=> ((RESPONSE ==
        CP_RC_UNALIGNED) or ((RESPONSE == CP_RC_WAIT)[*1:$] ##1
        RESPONSE == CP_RC_UNALIGNED));
endproperty
```

7.5 Registre procesoru Codix RISC

Pri overovaní registrov sa overujú tri základné požiadavky na čítanie a zápis. Operácie čítania vyžadujú, aby adresa, z ktorej sa bude čítať, neobsahovala žiadne nedefinované hodnoty, aby bola v rozmedzí pridelenom danému registru, a aby boli aj následne prečítané dáta plne známe. Formálne tvrdenia pokrývajúce tieto požiadavky boli implementované firmou Codasip a sú zobrazené v príklade 7.18. Analogicky sú formálne tvrdenia aplikované aj na operácie zápisu.

Príklad 7.18 Formálne tvrdenia overujúce čítanie a zápis do registrov

```
// Adresa, z ktorej sa má čítať nesmie obsahovať X hodnoty.
property regs_x_read_address;
   @(posedge CLK)
       !$isunknown(RE) && (RE == 1'b1) |-> !$isunknown(RA);
endproperty
// Hodnota adresy musí byť v danom rozmedzí
property regs_RA_range;
   @(posedge CLK)
       !$isunknown(RE) && (RE == 1'b1) && !$isunknown(RA) |->
       (RA>=MIN_ADDR) && (RA<=MAX_ADDR);
endproperty
// Ak sú splnené predošlé požiadavky, prečítané dáta neobsahujú X-hodnoty
property regs_no_x_read;
   @(posedge CLK)
      !$isunknown(RE) && (RE == 1'b1) && !$isunknown(RA) && ((RA>=MIN_ADDR)
      && (RA<=MAX_ADDR)) |-> !$isunknown(Q);
endproperty
```

7.6 Dekodér inštrukcií

Pri implementácii formálnych tvrdení pre dekodér inštrukcií je v prvom rade vykonaná kontrola na správnosť dát na portoch. Túto kontrolu zobrazuje príklad 7.20.

Príklad 7.19 Formálne tvrdenia overujúce správnosť dát dekodéru inštrukcií

```
// Signalizácia nevalidnej inštrukcie nesmie obsahovať X-ovú hodnotu
property asip_x_unknown_instruction;
    @(edge ACT, edge INVALID_INSTR)
        !$isunknown(ACT) && (ACT == 1'b1) |-> !$isunknown(INVALID_INSTR);
endproperty
// Indikácia výskytu neplatnej inštrukcie
property asip_unknown_instruction;
    @(edge ACT, edge INVALID_INSTR)
        !$isunknown(ACT) && !$isunknown(INVALID_INSTR) |->
        (INVALID_INSTR == 1'b0);
endproperty
```

Ďalšou kontrolovanou vlastnosťou dekodéru inštrukcií, ktorá bola určená ako kritická, je exkluzivita signálov rovnakých kategórií. Na zjednodušenie implementácie a zvýšenie znovupoužiteľnosti boli použité makrá overujúce exkluzivitu zobrazené v príklade 7.20. V tomto príklade je možné vidieť použitie daného makra, pričom vstupom mu je konkatenácia všetkých signálov danej skupiny, nad ktorou bola vykonaná operácia `or`.

Príklad 7.20 Formálne tvrdenia overujúce výlučný prístup v dekodéri inštrukcií

```
// Makro overujúce exkluzívny prístup
`define CODASIP_ABV_MUTEX_DRIVER( name, test_expr,
msg_id=ASSERTION_ERROR_MUTEX_DRIVER,
msg=,,More than one driver active for %0s.%0s.``, clock=CLK )
    property p_mutex_driver_``name;
        @(posedge clock)
        ##1 $onehot0(test_expr);
    endproperty
// Formálne tvrdenie konkatenujúce
`CODASIP_ABV_MUTEX_DRIVER( rd_mem_rw_D0, {
    |main_0_instr_hw_alu_imm_instr_hw_instr_hw_0_semantics_rd_mem_rw_D0,
    |main_0_instr_hw_alu_instr_hw_instr_hw_0_semantics_rd_mem_rw_D0,

⋮


    |main_0_instr_hw_system_wr_instr_hw_instr_hw_0_semantics_rd_mem_rw_D0})
```

7.7 Evaluácia formálnych tvrdení v simulácii

Súčasťou vygenerovaného verifikačného prostredia sú predpripravené skripty v jazyku TCL, pomocou ktorých je priamo spustená simulácia v programe QuestaSim, resp. Modelsim. a skripty, ktoré je možné interpretovať priamo vo vybranom simulačnom nástroji. Po štandardnom spustení simulácie je spustená verifikácia procesora so zvolenou aplikáciou a do terminálu sú vypisované aktuálne hlásenia o jednotlivých formálnych tvrdeniach. Po ukončení verifikácie je možné vidieť súhrnný výpis o dokončenej verifikácii s informáciami (obrázok 12) ako sú:

- celkový počet porušení formálnych tvrdení (`UVM_ERROR`),
- celkový počet ostatných hlásení o formálnych tvrdeniach (`UVM_INFO` a `UVM_WARNING`),
- celkový počet fatálnych problémov (`UVM_FATAL`),
- počet porušení formálnych tvrdení v jednotlivých komponentoch (`ASSERTION_ERROR_CLB` a `ASSERTION_ERROR_REGS`),
- počet verifikovaných zariadení, počet overovaní, počet cyklov, celkový čas trvania a výsledný status.

```
# --- UVM Report catcher Summary ---
#
#
# Number of demoted UVM_FATAL reports  :    0
# Number of demoted UVM_ERROR reports  :    0
# Number of demoted UVM_WARNING reports:    0
# Number of caught UVM_FATAL reports   :    0
# Number of caught UVM_ERROR reports   :    0
# Number of caught UVM_WARNING reports :    0
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   11
# UVM_WARNING :    0
# UVM_ERROR :  296
# UVM_FATAL :    0
# ** Report counts by id
# [ASSERTION_ERROR_CLB]   247
# [ASSERTION_ERROR_REGS]    49
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [dut:]     1
# [gold:]     1
# [loader:build:]     1
# [report_phase:user_status]     4
# ** Note: $finish    : c:/questa_sim64_10.1d/win64/.
#    Time: 1280 ns  Iteration: 58  Instance: /top
```

Obrázok 12: Výpis súhrnných informácií o ukončenej simulácii

Prehľad overovaných formálnych tvrdení je možné nájsť v samostatnom okne simulátora QuestaSim nazvanom „Assertions“ (obrázok 13), ktoré zobrazuje všetky formálne tvrdenia načítané vo verifikačnom prostredí (Name), komponent verifikačného prostredia, v ktorom sa nachádzajú (Design Unit), ich presné znenie (Assertion Expression) a informácie o každom z nich zvlášť. Z obrázku možeme vidieť aj celkový počet pokusov o vyhodnotenie formálneho tvrdenia (Cummulative Threads), počet úspešných (Pass Count) a neúspešných (Failure Count) vyhodnotení, maximálny počet súčasných vyhodnocovaní jedného formálneho tvrdenia (Peak Active Count), či nastavenie zberu informácií pre dané formálne tvrdenia (Pass/Failure Log).

Úpravou vygenerovaných spúšťacích skriptov je možné oproti štandardnému zobrazovaniu docieliť grafickú reprezentáciu výsledkov funkčnej verifikácie a zobraziť priebeh overovaných formálnych tvrdení (obrázok 14). Na dosiahnutie popísaného zobrazenia je potrebné aplikovať nasledujúce kroky:

1. nástroj QuestaSim musí byť spúšťavý s parametrom `voptargs`, ktorý musí obsahovať voľbu `a` (napríklad `-voptargs=,,+acc=arn‘‘`),

2. taktiež je potrebné pridať parameter `-assertdebug` do príkazu apúšťajúceho simuláciu,

Name	Design Unit
assert__memory_no_x_read	codasip_memory_read_assertions
assert__regs_initialized_before_read	codix_risc_ca_core_regs_t_abv_probe
assert__regs_no_x_read	codasip_regs_no_x_read_assertion
assert__regs_no_x_read	codasip_regs_no_x_read_assertion
assert__regs_no_x_read	codasip_regs_no_x_read_assertion
assert__memory_read_ack	codasip_memory_read_assertions
assert__clb_wrong_resp_on_correct_request_ready	codasip_clb_assertions
assert__clb_wrong_resp_on_correct_request_ready	codasip_clb_assertions
assert__memory_request_not_processed	codasip_memory_assertions
assert__regs_initialized_before_read	codix_risc_ca_core_regs_t_abv_probe
assert__regs_initialized_before_read	codix_risc_ca_core_regs_t_abv_probe

...

Assertion Expression	Pass Log	Failure Log
assert(@(posedge CLK) (IFRESPONSE==1)|->~$isunknown(Q))	on	-
assert(@(posedge CLK) ((WE0==0&&WA0==0) ##[0:$](((RE0==1&&RA0==0)||(RE1==1&&RA1=...	on	-
assert(@(posedge CLK) (RE==1)|->~$isunknown(Q))	on	-
assert(@(posedge CLK) (RE==1)|->~$isunknown(Q))	on	-
assert(@(posedge CLK) (RE==1)|->~$isunknown(Q))	on	-
assert(@(posedge CLK) ((((SUPPORTED&&REQUEST==1)&&STATUS==1)&&SI+SC<=4) ##[1]1)|-...	on	-
assert(@(posedge CLK) ((SUPPORTED&&STATUS==1))|=>((RESPONSE>=1&&RESPONSE<=5)&&(...	on	-
assert(@(posedge CLK) ((SUPPORTED&&STATUS==1))|=>((RESPONSE>=1&&RESPONSE<=5)&&(...	on	-
assert(@(posedge CLK) (((SUPPORTED&&REQUEST==2)||(REQUEST==1&&STATUS==1)))|->(1 #...	on	-
assert(@(posedge CLK) ((WE0==0&&WA0==28) ##[0:$](((RE0==1&&RA0==28)||(RE1==1&&RA...	on	-
assert(@(posedge CLK) ((WE0==0&&WA0==29) ##[0:$](((RE0==1&&RA0==29)||(RE1==1&&RA...	on	-

...

Failure Count ▽	Pass Count	Active Count	Vacuous Count	Attempt Count	Peak Active Count	Memory	Cumulative Threads
103	0	0	24	127	1	0B	103
35	0	0	92	127	15	0B	135
20	77	0	30	127	1	0B	97
15	82	0	30	127	1	0B	97
8	89	0	30	127	1	0B	97
6	103	1	17	127	2	80B	110
6	103	1	17	127	2	80B	110
6	103	1	17	127	2	80B	110
6	103	1	17	127	2	320B..	562
0	0	0	127	127	0	0B	0
0	0	0	127	127	0	0B	0

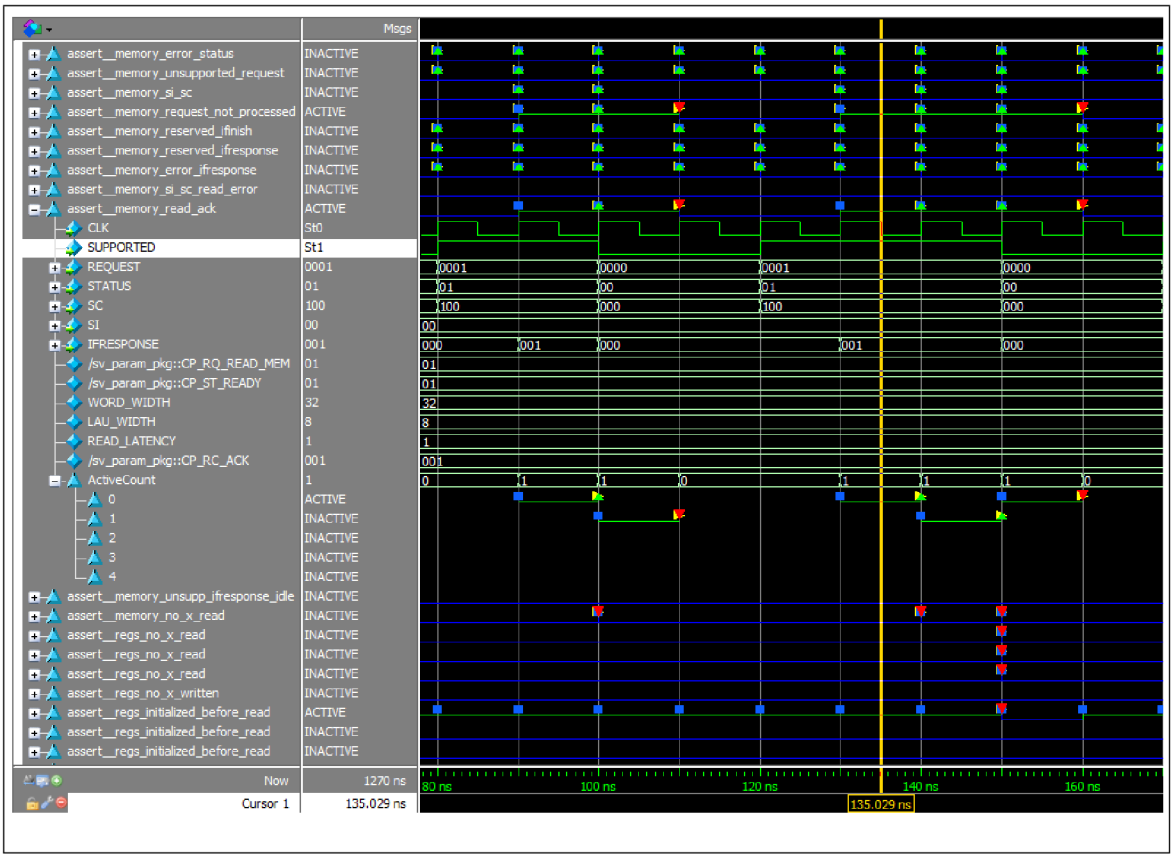
Obrázok 13: Okno Assertions zobrazujúce všetky formálne tvrdenia

3. v nástroji QuestaSim je po načítaní simulačného prostredia a pred spustením samotnej simulácie (príkaz `run -all`) potrebné označiť okno `Assertions` a následne odznačiť prípadné vybrané formálne tvrdenia `Edit->Unselect All` (musí byť vybrané okno s formálnymi tvrdeniami a zároveň nesmie byť označené žiadne formálne tvrdenie),

4. v kontextovom menu pre formálne tvrdenia (okno `Assertions`) je potrebné zapnúť logovanie (`Assertions -> Configure -> Failure/Passes -> Logging:on`),

5. je potrebné pridať všetky formálne tvrdenia z načítaného prostredia do okna `Wave` (označenie všetkých assertionov a následne `Add -> To Wave`),

6. následne je možné spustiť samotnú simuláciu `run -all`.



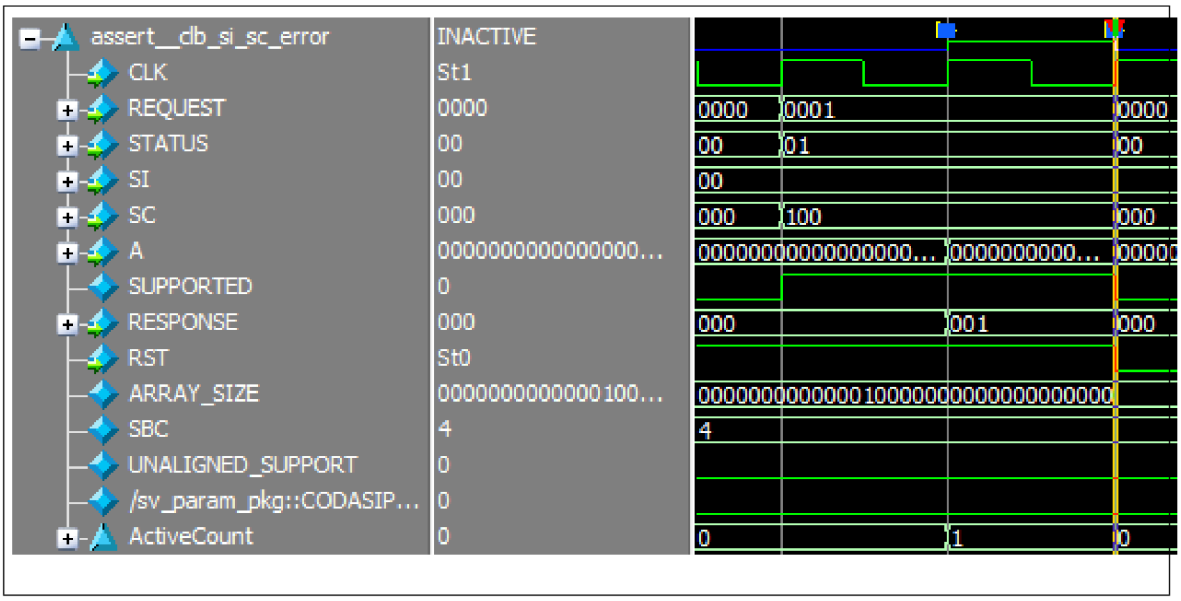
Obrázok 14: Grafické zobrazenie overovania formálnych tvrdení v čase

Nakoľko sa v okne `Assertions` (obrázok 13) neuvádzajú žiadne časové údaje o vyhodnoteniach daných formálnych tvrdení a vo veľkých projektoch je pomerne pracné spätne dohľadávať tieto informácie vo výpisoch, je možné považovať túto možnosť grafickej reprezentácie za kľúčovú v procese ladenia komplexných návrhov. Jednotlivé overovania formálnych tvrdení (obrázok 14), ich priebeh v čase a aj výsledky vyhodnocovania sú zobrazené ako:

- modrý štvorec (začiatok vyhodnocovania),
- zelený trojuholník (úspešná evaluácia),
- červený trojuholník (neúspešná evaluácia),
- žltý trojuholník (zhoda s predchádzajúcou evaluáciou),
- modrá čiara (neaktívne formálne tvrdenie),

- zelená čiara (aktuálne vyhodnocovanie).

Prvým krokom v procese ladenia formálnych tvrdení po úspešnom spustení verifikácie popísanej vyššie, je vyhľadanie chybných evaluácií týchto formálnych tvrdení. Pomocou zobrazenia všetkých formálnych tvrdení a ich vyhľadaní v okne *Wave* je možné nájsť presný čas a hodnoty daných signálov v dobe evaluácie formálneho tvrdenia. Na obrázku 15 je vidieť formálne tvrdenie, ktorého vyhodnotenie je takmer vždy neúspešné, čo často indikuje chybu vo formálnom tvrdení samotnom. Ďalšou možnosťou ponúkanou simulačným nástrojom je zobrazenie *Dataflow* okna, ktoré umožňuje pohľad na prepojenie celkov a funkčných blokov (obrázok 16).

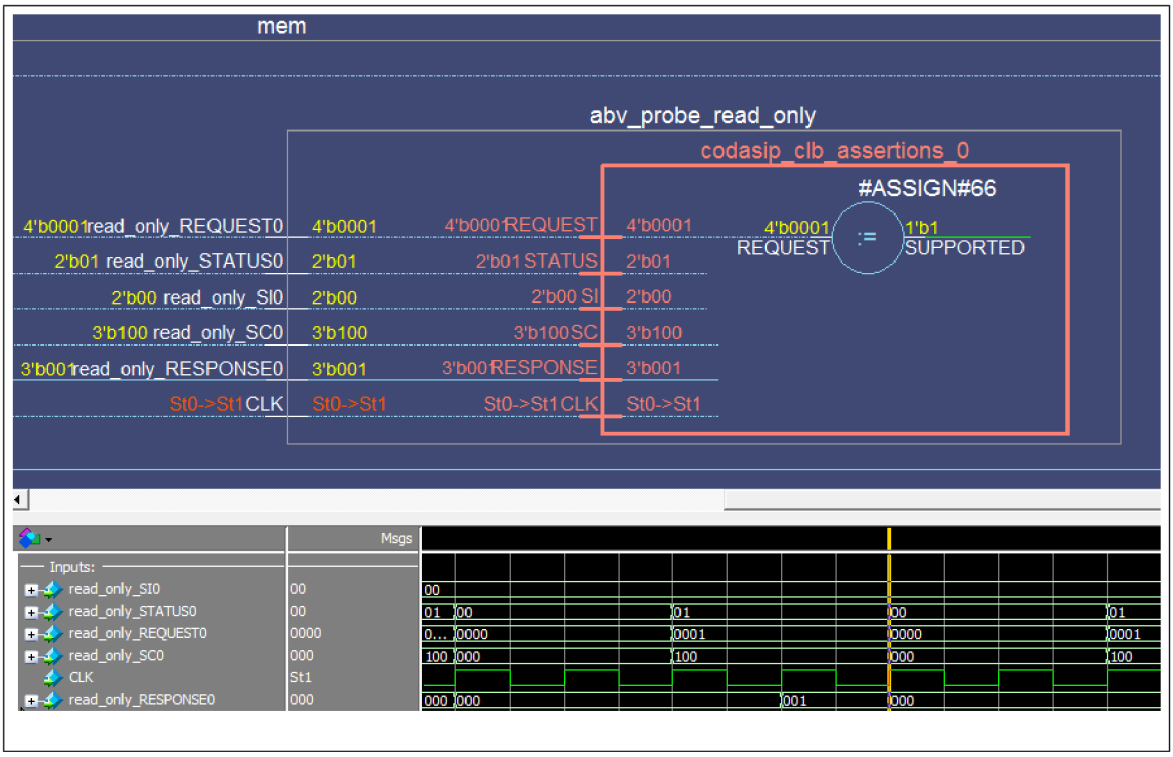


Obrázok 15: Signály v danom formálnom tvrdení

Pri pohľade na stav signálov formálneho tvrdenia vidíme, že pri požiadavke na čítanie celého bloku dát je hodnota signálu `SC` predstavujúceho počet požadovaných pod-blokov maximálna možná (v našom prípade štyri). Hodnota signálu `SI` predstavujúca počiatočný pod-blok, od ktorého sa začne čítanie, je rovná nule. V tomto prípade ide o validnú požiadavku na čítanie, v ktorej má byť prečítaný celý blok dát. Ak je počet pod-blokov v bloku `SBC`, (anglicky Sub-Block Count) rovný počtu požadovaných pod-blokov, musí byť počiatočný index rovný nule a v tom prípade sa `SBC = SC + SI = SC + 0`. Po otvorení zdrojvého kódu formálneho tvrdenia je možné zistiť nesúlad s návrhom, kde podmienka v kóde obmedzuje platné požiadavky čítania `SI + SC < SBC`, čím nepovoluje čítanie celého bloku dát. Po upravení príslušnej podmienky v kóde a opätovnom spustení simulácie už nebolo dané formálne tvrdenie ani raz vyhodnotené ako neplatné.

7.8 Pokrytie formálnych tvrdení

Hlavnou úlohou pokrytia formálnych tvrdení daného návrhu je zistiť, do akej miery je tento návrh otestovaný. Bežnou praxou je využívanie rôznych nástrojov poskytujúcich informácie o pokrytí testovaných vstupov, či sa jedná o black-box alebo white-box testovanie. V práci je využitý vyššie popísaný nástoj QuestaSim, ktorého verifikačný manažment pracuje nad jednotnou databázou pokrytí (anglicky Unified Coverage Database, skrátene UCDB). Táto



Obrázok 16: Prepojenie blokov v pohľade Dataflow

databáza zachytáva akýkoľvek zdroj pokrývacích dát generovaných rôznymi verifikčnými nástrojmi a následne ich spracováva. QuestaSim natívne využíva tento formát na ukladanie dát o pokrytí kódu, funkčnom pokrytí a formálnych tvrdeniach.

Upravením spúšťacích skriptov, konkrétne pridaním parametrov `-cvg -assert -directive` do príkazu `coverage save`, je možné docieliť ukladanie dát z ukončenej simulácie do UCDB formátu. Pokrytie pomocou formálnych tvrdení bolo otestované sadou štyridsiatich náhodne vybraných aplikácií, pričom verifikácia bola spustená zvlášť pre každú aplikáciu. Po ukončení verifikácie s využitím sady aplikácií sú výsledky jednotlivých spustení uložené v UCDB súbore v zložke obsahujúcej vstupné aplikácie. Výsledky jednotlivých verifikácií je možné zobraziť príkazom `vcover report <voľby> <vstupný súbor>`.

Získavanou hodnotou je percentuálne pokrytie formálnych tvrdení. V aktuálnej implementácii verifikačného prostredia nie sú využívané žiadne formálne tvrdenia na overovanie pokrytia, a preto nemá zmysel skúmať rozdiely v pokrytí pred a po doplnení týchto formálnych tvrdení. Pri formálnych tvrdeniach sa počíta percentuálny podiel overovaných a tých, ktoré boli reálne overené. Získané hodnoty sú uvedené v tabuľke 7.1 (stĺpec pokrytie formálnych tvrdení) pre jednotlivé aplikácie, resp. verifikácie spustené s týmito aplikáciami. Ďalšou získavanou hodnotou je percentuálna úspešnosť všetkých formálnych tvrdení (stĺpec úspešnosť formálnych tvrdení) vyjadrujúca percentuálny podiel tvrdení, ktoré nikdy nezlyhali, a zároveň aspoň raz boli úspešne vyhodnotené. Celkové vyhodnotenie všetkých aplikácií bolo uskutočnené pomocou príkazu `vcover -merge`, ktorý pri počítaní výsledných hodnôt využíva UCDB dáta jednotlivých výstupov a zlučuje ich do výsledného UCDB úložiska, z ktorého sa počítajú výsledné hodnoty.

Z tabuľky 7.1 je možné vidieť, že hodnota nameraných dát sa nemení s meniacou sa aplikáciou. Z toho je možné usúdiť, že formálne tvrdenia, ktoré sú neúspešne vyhodnocované v jednej aplikácii, budú pravdepodobne neúspešne vyhodnotené aj v ostatných aplikáci-

Tabulka 7.1: Pomer overených formálnych tvrdení v danej sonde

Aplikácia	Pokrytie formálnych tvrdení	Úspešnosť formálnych tvrdení	Funkčné pokrytie
compare-3	73,2%	96,6%	41,1%
enum1	73,2%	96,6%	41,7%
conversion	73,2%	96,6%	41,9%
charconst-4	73,2%	96,6%	41,2%
fp-cmp-7	73,2%	96,6%	41,2%
loop-9	73,2%	96,6%	41,3%
divcmp-5	73,2%	96,6%	41,7%
⋮			
mayalias-2	73,2%	96,6%	41,3%
Zlúčené	73,2%	96,6%	41,7%

ách. Testovacia sada totiž nie je špeciálne zameraná na overovanie hraničných podmienok testovaných komponentov. Tieto komponenty sú overované na úrovni signálov a nie na úrovni príkazov v programe, a preto je možné usúdiť, že aj značne odlišné aplikácie sa preložia do podobnej sady príkazov a sú interpretované podobnými sledmi hodnôt jednotlivých signálov. Meniace sa hodnoty v poslednom stĺpci naopak zobrazujú funkčné pokrytie pomocou skupín pokrytia a bodov pokrytia, ktoré reflektujú využívanie jednotlivých kombinácií vstupných hodnôt líšiacich sa pre jednotlivé aplikácie. Tento typ funkčného pokrytia nevyužíva formálne tvrdenia a nevyplýva zo zadania, no je považovaný za podstatnú časť celkového overovania pokrytia, a preto je táto časť bližšie popísaná v prílohe.

Kapitola 8

Záver

Cieľom práce bolo oboznámenie sa s frameworkom od firmy Codasip, štúdium techník a postupov vytvárania verifikačných prostredí v jazyku SystemVerilog ako aj techník verifikácie založenej na formálnych tvrdeniach. Hlavným cieľom bolo navrhnúť koncept pre overovanie správnosti týchto procesorov pomocou verifikácie založenej na formálnych tvrdeniach, implementovať ho v jazyko SystemVerilog Assertions a otestovať ho na vybranom procesore.

Po preštudovaní techník verifikácie založenej na formálnych tvrdeniach a oboznámení sa s jazykom SystemVerilog Assertions a jeho možnými aplikáciami na procesory s aplikačne-špecifickou inštrukčnou sadou bolo navrhnuté a následne implementované riešenie overovania správnosti procesoru Codix RISC. Riešenie spočívalo v identifikovaní kľúčových miest vhodných pre tento druh verifikácie. Pomocou nástrojov firmy Codasip bolo vygenerované verifikačné prostredie v jazyku SystemVerilog, ktoré bolo využité ako základ pre implementáciu formálnych tvrdení v jazyku SystemVerilog Assertions.

V rámci práce bola v spolupráci s firmou Codasip implementovaná rozsiahla sada formálnych tvrdení pre daný procesor, ktorá bola následne otestovaná. Táto sada bola navrhnutá s ohľadom na znovupoužiteľnosť pri práci s ďalšími procesormi, a preto boli jednotlivé skupiny formálnych tvrdení rozdelené do modulov podľa nadväznosti na základné komponenty procesorov (napr. pamäť, registre).

Možným pokračovaním práce by mohlo byť rozšírenie o ďalšie moduly a periférie nenachádzajúce sa priamo v zvolenom procesore, ale ktoré sú štandardne k procesoru pripájané, ako napríklad cache, jednotka pre správu prerušení, jednotka pre správu pamäte a iné. Ďalšie možné pokračovanie je implementácia pomocou iných jazykov ako napríklad PSL.

Literatura

[1] Bergeron, J.; Cerny, E.; Hunter, A.; aj.: *Verification Methodology Manual for SystemVerilog*. Springer US, 2006, ISBN 9780387255569.

[2] Cerny, E.; Dudani, S.; Havlicek, J.; aj.: *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing, 2014, ISBN 9783319071398.

[3] Chen, X.; Luo, Y.; Hsieh, H.; aj.: Assertion Based Verification and Analysis of Network Processor Architectures. *Design Automation for Embedded Systems*, ročník 9, č. 3, 2004: s. 163–176, ISSN 0929-5585, doi:10.1007/s10617-005-1193-5.

[4] Faisal Haque, K. K., Jonathan MMichelson: *The Art of Verification with SystemVerilog Assertions*. Verification Central, 2006, ISBN 9780971199415.

[5] Foster, H. D.; Krolnik, A. C.: *Creating Assertion-Based IP*. Springer Science+Business Media, LLC, 2008, iSBN 978-0-387-36641-8.

[6] Foster, H. D.; Krolnik, A. C.; Lacey, D. J.: *Assertion-Based Design*. Springer Science+Business Media, LLC, druhé vydání, 2004, iSBN 1-4020-8027-1.

[7] Krygowski, C. A.; Bair, D. G.; Gott, R. M.; aj.: Functional verification of the IBM System z10 processor chipset. *IBM Journal of Research and Development*, ročník 53, č. 1, 2009: str. 3.

[8] Kulkarni, M.; Bommi J., B.: Assertion-Based Verification for the SpaceCAKE Multiprocessor - A Case Study. In *Hardware and Software, Verification and Testing*, *Lecture Notes in Computer Science*, ročník 3875, editace S. Ur; E. Bin; Y. Wolfsthal, Springer Berlin Heidelberg, 2006, ISBN 978-3-540-32604-5, s. 43–55, doi:10.1007/11678779˙4.

[9] Lepenica, N.: *Assertion Based Verification on Senior DSP*. Diplomová práce, Linköping University, 2011.

[10] Liu, D.: *Embedded DSP processor design: application specific instruction set processors*, ročník 2. Morgan Kaufmann, 2008, iSBN 978-0-12-374123-3.

[11] Packiaraj, V.: *Study, Design and Implementation of an Application Specific Instruction Set processor for Specific DSP Task*. Diplomová práce, Linköping Institute of Technology, 2008.

[12] Rameshlal, D. V.; Kumar, N.: Embeded DSP processor design using CoWare processor designer and Magma layout tool. 2010.

[13] Spear, C.: *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2008, ISBN 9780387765303.

[14] WWW stránky: 1012-2012 - IEEE Standard for System and Software Verification and Validation. Dostupné na https://standards.ieee.org/findstds/standard/1012-2012.html.

[15] WWW stránky: Assertion-Based Verification. Dostupné na http://www.mentor.com/products/fv/methodologies/abv/.

[16] WWW stránky: Codasip Framework. Dostupné na http://www.codasip.com.

[17] WWW stránky: LLVM Project. Dostupné na http://llvm.org/.

[18] WWW stránky: .NET Micro Framework. Dostupné na http://en.wikipedia.org/wiki/.NET_Micro_Framework.

[19] WWW stránky: QuestaSim. Dostupné na http://www.mentor.com/products/fv/questa/.

[20] Šimková, M.; Kajan, M.: Verifikace číslicových obvodů. Dostupné na http://www.fit.vutbr.cz/~isimkova/PCS_presentation/pcs2012.pdf, 2012.

[21] Šimková, M.; Kajan, M.: Verifikace číslicových obvodů: využití metod a nástrojů formální verifikace. http://www.fit.vutbr.cz/~isimkova/grants.php?file=/proj/650/abv-long.pdf&id=650, 2013.

Přílohy

Příloha A

Funkčné pokrytie bez použitia formálnych tvrdení

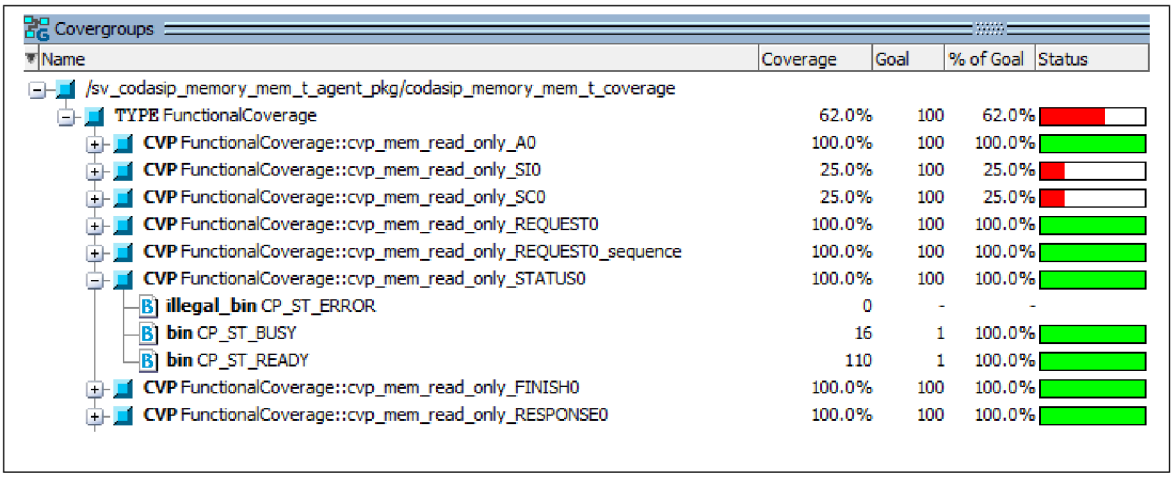
Funkčné pokrytie je možné dosiahnuť aj bez využitia formálnych tvrdení pomocou nižšie popísaného prístupu. Napriek jeho značnému prínosu a veľkému využitiu, tento prístup nie je obsiahnutý v zadaní práce, ani z jej popisu nijako nevyplýva. Kvôli týmto dôvodom je popis nasledujúcej metódy a jej realizácie vyňatý z práce samotnej a je uvedený v samostatnej prílohe nadväzujúcej na funkčné pokrytie využívajúce formálne tvrdenia uvádzané v práci.

Vygenerované verifikačné prostredie pre procesor Codix RISC v sebe obsahuje niekoľko agentov, z ktorých každý obsahuje predpripravené triedy pokrytia. V každej z týchto tried pokrytia sú pri generovaní vytvorené základné body pokrytia (anglicky coverpoint) pokrývajúce vstupné a výstupné signály. Coverpoint je štruktúra, ktorá obsahuje množinu košov (anglicky bins), prípadne ďalšie coverpointy, v ktorých sú vstupné hodnoty rozdelené do príslušných košov definovaných manuálne, prípadne automaticky na základe pravidiel vymedzujúcich tieto koše. Okrem coverpointov sú v triede implementované aj tzv. krížové pokrytia využívajúce existujúce coverpointy ako svoje vstupy a svoje koše vymedzujú použitím kombinácií a obmedzení vstupných coverpointov a ich košov. Všetky popísané coverpointy a krížové pokrytia v danej triede sú zastrešené v jednej skupine porkytia (anglicky covergroup). Covergroup obsahuje okrem všetkých vyššie popísaných vecí aj hodinové udalosti a nastavenia danej skupiny.

Pomocou coverpointov pokrývajúcich jednotlivé vstupy vo všetkých agentoch získame prehľad o tom, ako dôkladne dané vstupné dáta preverujú všetky možné kombinácie vstupov. Výstupom je súhrn (viď obrázok 17) zobrazujúci výsledky pre jednotlivé skupiny pokrytia, konkrétne coverpointy a zobrazenie pomerového rozdelenie vstupov v rámci coverpointu do daných košov.

Príklad 17 znázorňuje pokrytie jedného vstupného portu agenta pracujúceho s pamäťou. Coverpointy pre jednotlivé vstupy boli doimplementované firmou Codasip, resp. v aktuálnej verzii Codasip frameworku sú automaticky generované do verifikačného prostredia.

Obdobným spôsobom sú generované aj niektoré krížové pokrytia, ako napríklad vzťah statusu a požiadavky (viď príklad A.2) pri práci s pamäťou v režime read-only, kedy sú ako platné hodnoty statusu a požiadavky evidované len hodnoty zaradené do košov daných coverpointov (neplatné a ignorované koše v coverpointoch samotných nie sú brané do úvahy). Následne sa v krížovom overovaní dodatočne odfiltrujú také kombinácie košov, ktoré obsahujú špecifikovaný kôš coverpointu.



Obrázok 17: Skupina pokrytí pre prácu s pamäťou a zobrazenie coverpointov a jednotlivých košov

Príklad A.1 Coverpoint pokrývajúci vstupný port

```
// Požadovaný počet pod-blokov z daného bloku čítaných dát
cvp_mem_read_write_SC0 : coverpoint m_transaction_h.read_write_SC0
 iff( !$isunknown(m_transaction_h.read_write_SC0) ) {
    // legálne hodnoty - vytvorenie košov pre čísla 1 až 4
    bins sbc[]                  = {[1:4]};
    // ignorovanie nepodstatných hodnôt
    ignore_bins zero_access     = {0};
    // zakázané hodnoty - všetky ostatné
    illegal_bins illegal_sbc[] = default;
    // nastavenia coverpointu
    option.weight = 0; }
```

Príklad A.2 Krížové pokrytie

```
// Krížové overovanie statusu a požiadavky
cvp_mem_read_only_REQUEST0_ready: cross cvp_mem_read_only_STATUS0,
                                  cvp_mem_read_only_REQUEST0 {
    // Ignorovanie kombinácií košov, kde sa vyskytuje kôš statusu
    // s hodnotou BUSY indikujúcou nepripravenosť komponentu
    ignore_bins status_not_ready =
                binsof(cvp_mem_read_only_STATUS0.CP_ST_BUSY); }
```

Krížové pokrytie je vhodným prostriedkom na získanie štatistických údajov ako je pravdepodobnosť odpovede `zariadenie je zaneprázdnené` pri pokuse o čítanie (viď príklad A.3). Keďže existujú len tri platné odpovede a pre každú odpoveď existuje samostatný kôš, percentuálne rozdelenie medzi jednotlivé koše nám poskytne približnú informáciu o pravdepodobnostiach odpovede na požiadavku čítania. Obdobne zapísané je aj pokrytie pre prístup do pamäte s prístupom na čítanie aj zápis, no v tomto prípade sú výsledné koše duplikované pre operáciu čítania aj zápisu.

Príklad A.3 Krížové pokrytie pokrývajúce odpovede na požiadavku čítania

```
// Pokrytie odpovedí na operáciu čítania
cvp_mem_read_only_responses: cross cvp_mem_read_only_STATUS0,
    cvp_mem_read_only_REQUEST0, cvp_mem_read_only_RESPONSE0,
    cvp_mem_read_only_RESPONSE0_seq_raise_resp {
    // Odpoveď WAIT
    bins read_operation_wait = binsof(
        cvp_mem_read_only_RESPONSE0_seq_raise_resp.raise_wait);
    // Odpoveď ACK
    bins read_operation_ack = binsof(
        cvp_mem_read_only_RESPONSE0_seq_raise_resp.raise_ack);
    // Odpoveď IDLE
    bins read_operation_idle = binsof(
        cvp_mem_read_only_RESPONSE0_seq_raise_resp.raise_idle);
    // Zariadenie nie je pripravené
    ignore_bins status_not_ready   = binsof(
        cvp_mem_read_only_STATUS0.CP_ST_BUSY);
    // Žiadna požiadavka nebola zadaná
    ignore_bins no_operation       = binsof(
        cvp_mem_read_only_REQUEST0.CP_RQ_NONE); }
```

V opise procesoru Codix RISC bolo, ako jedna z jeho výhod, spomenuté ponechanie hazardov na prekladačoch s výnimkou minutia dátovej cache a čakania na pamäťovú zbernicu. Pokiaľ však ide o prácu s registrami, je potrebné pokryť a podchytiť aj takéto hraničné prípady. Pri prístupe do registrov však môže zápis aj čítanie do/z jedného pamäťového miesta nastať naraz, prípadne v krátkom časovom slede, čo potenciálne vytvára priestor pre rôzne chyby a nekonzistenciu dát. Pri implementácii pokrytia týchto prípadov sme využili oba vyššie spomínané prístupy. Prvým prístupom (viď príklad A.4) overujeme čítanie a zápis nastvajúce v rovnakom takte.

Pri implementácii pokrytia čítania a zápisu, ktoré nenastávajú simultánne, ale v krátkom slede (v našom prípade v nasledujúcom takte) už nepostačuje predchádzajúci prístup a opäť je potrebné využiť formálne tvrdenia s direktívou `cover` (viď príklad A.5). Pri implementácii pokrytia pomocou formálnych tvrdení je možné využiť všetky už existujúce formálne tvvrdenia existujúce v návrhu.

Výsledky, ktoré implementácia funkčného pokrytia priniesla, môžeme rozdeliť do dvoch skupín podľa prístupu, ktorý bol pri ich implementácii aplikovaný. V prvom prípade boli využité pripravené skupiny pokrytí, do ktorých boli v práci doplnené ďalšie. Testovanie prebiehalo použitím aplikácií pre daný procesor, ktoré boli spolu s modelom procesoru poskytnuté firmou Codasip. Sledovanou hodnotou je najmä priemerné percentuálne pokrytie jednotlivých vstupných portov pomocou coverpointov a ich kombinácií prostredníctvom krížového pokrytia. Získané hodnoty je možné vidieť v tabuľke A.1, presnejšie v stĺpci: Funkčné pokrytie.

Na meranie rozdielu celkového pokrytia boli odstránené naprogramované krížové pokrytia a coverpointy a opäť bolo vykonané meranie pokrytia rovnakou metódou s výsledkom celkového funkčného pokrytia s využitím skupín pokrytia 41, 4%. Pri porovnávaní výsledkov je však potrebné brať do úvahy, že niektoré coverpointy boli naprogramované, aby slúžili

Príklad A.4 Súčasné čítanie a zápis z/do jedného pamäťového miesta

```
// Prístup pre čítanie aj zápis
cvp_regs_read_and_write_RE0: cross cvp_regs_WE0, cvp_regs_RE0,
                    cvp_regs_WA0_full, cvp_regs_RA0_full {
    // Obe operácie musia byť aktívne naraz
    ignore_bins disabled1 = binsof(cvp_regs_WE0.disabled) ||
                    binsof(cvp_regs_RE0.disabled);
    // Adresy pre čítanie a zápis sa musia zhodovať
    bins read_0_write_0_same_address_parallel =
                    binsof(cvp_regs_RE0.enabled) &&
                    binsof(cvp_regs_WE0.enabled) &&
                    binsof(cvp_regs_RA0_full) intersect {waddr} &&
                    binsof(cvp_regs_WA0_full) intersect {waddr};
    // Ignorovanie ostatných prípadov
    ignore_bins simultanious_read_and_write_different_addr =
                    binsof(cvp_regs_RE0.enabled) &&
                    binsof(cvp_regs_WE0.enabled); }
```

Príklad A.5 Pokrytie sledu operácií `read after write`

```
// Zápis nasledovaný čítaním z rovnakej adresy
property codix_regs_RAW_operations;
    @(posedge CLK)
        !$isunknown(WE0) && (WE0 == 1'b1) && !$isunknown(WA0) |=>
        !$isunknown(RE0) && (RE0 == 1'b1) && !$isunknown(RA0) &&
        (RA0 == $past(WA0)) |-> 0;
endproperty
cover property (codix_regs_RAW_operations);
```

Tabulka A.1: Pomer overených formálnych tvrdení v danej sonde

Aplikácia	Pokrytie formálnych tvrdení	Úspešnosť formálnych tvrdení	Funkčné pokrytie
compare-3	73,2%	96,6%	41,1%
enum1	73,2%	96,6%	41,7%
conversion	73,2%	96,6%	41,9%
charconst-4	73,2%	96,6%	41,2%
fp-cmp-7	73,2%	96,6%	41,2%
loop-9	73,2%	96,6%	41,3%
divcmp-5	73,2%	96,6%	41,7%
⋮			
mayalias-2	73,2%	96,6%	41,3%
Zlúčené	73,2%	96,6%	41,7%

výhradne ako vstupy pre krížové pokrytia. Cieľom niektorých pokrytí nebolo zobrazovať bežne sa vyskytujúce dáta, ale slúžili iba na zachytávanie hraničných prípadov. Príkladom je kontrola „out-of-range“ adresy pri práci s pamäťou. Pri tomto krížovom pokrytí je ako vstup coverpoint porkývajúci vstupnú adresu. Jeho platnými košmi sú hodnoty klasifikované ako „out-of-range“ a slúži ako komplement k pôvodnému coverpointu adresy, aby zachytil neplatné hodnoty.

V testovacej sade aplikácií sa spomínaná hodnota nevyskytla ani raz, a preto je celkové percentuálne pokrytie znížené oproti stavu, kedy sa tieto hraničné hodnoty nekontrolovali. Okrem krížového pokrytia s nula-percentným pokrytím sú nulové aj jeho vstupné coverpointy zachytávajúce hraničné hodnoty. Práve kvôli takémuto viacnásobnému zarátavaniu nulových hodnôt dochádza k percentuálnemu poklesu pokrytia, napriek tomu, že počet overovaných prípadov sa zvýšil.

Příloha B

Obsah CD

Priložené CD obsahuje:

- adresár `src` obsahujúci zdrojové kódy,
- súbor `README.txt` obsahujúci krátky návod na spustenie verifikácie, zakomponovanie iplementovaných zdrojových kódov apod. ,
- súbor PDF obsahujúci technickú správu k diplomovej práci vo verzii pre informačný systém.