VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

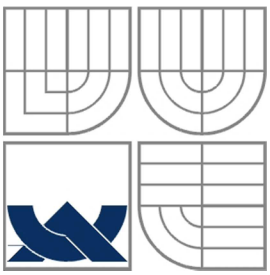
FUNKČNÁ VERIFIKÁCIA VÝPOČTOVÝCH JEDNOTIEK PROCESORU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

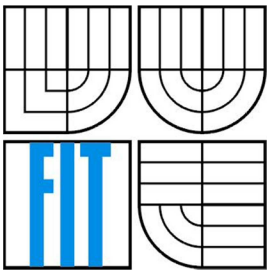
AUTOR PRÁCE
AUTHOR

LUKÁŠ VALACH

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FUNKČNÍ VERIFIKACE VÝPOČETNÍCH JEDNOTEK PROCESORU

FUNCTIONAL VERIFICATION OF PROCESSOR EXECUTION UNITS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ VALACH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2012

Abstrakt

Práce se zaobírá začleněním procesu funkční verifikace do vývojového cyklu návrhu funkčních jednotek v prostředí pro souběžný návrh hardwaru a softwaru systému Codasip. Cílem bylo navrhnout a implementovat verifikační prostředí v jazyku SystemVerilog pro verifikaci automaticky generované hardwarové reprezentace těchto jednotek. Na začátku jsou rozebrány přínosy a obvyklé postupy při funkční verifikaci a vlastnosti systému Codasip. Dále je v práci popsán návrh, implementace, analýza průběhu a výsledků testů verifikace simulačního modelu aritmeticko-logické jednotky. Závěrem jsou zhodnoceny dosažené výsledky práce a navrhnuta zlepšení pro možný další rozvoj verifikačního prostředí.

Abstract

The thesis deals with integration of functional verification into the design cycle of execution units in a hardware-software co-design environment of the Codasip system. The aim of the thesis is to design and implement a verification environment in SystemVerilog in order to verify automatically generated hardware representation of the execution units. In the introduction, advantages and basic methods of functional verification and principles of the Codasip system are discussed. Next chapters describe the process of design and implementation of the verification environment of arithmetic-logic unit as well as the analysis of the results of verification. In the end, a review of accomplished goals and the suggestions for future development of the verification environment are made.

Klíčová slova

funkční verifikace, Codasip, SystemVerilog, analýza pokrytí

Keywords

functional verification, Codasip, SystemVerilog, coverage analysis

Citace

Valach Lukáš: Funkčná verifikácia výpočtových jednotiek procesoru, bakalářská práce, Brno, FIT VUT v Brně, 2012

Funkčná verifikácia výpočtových jednotiek procesoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Karla Masaříka, Ph.D. Další informace mi poskytli Ing. Marcela Šimková a Ing. Zdeněk Přikryl, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Valach
16. 5. 2012

Poděkování

Děkuji vedoucímu práce Ing. Karlovi Masaříkovi, Ph.D. za lidský přístup, Ing. Zdeňkovi Přikrylovi, Ph.D. za ochotu a promptní řešení problémů a především Ing. Marcele Šimkovej za odbornou pomoc, ochotu při konzultacích a trpělivost při termínech.

Obsah

1 Úvod

V priebehu času získali počítačové systémy dôležité postavenie v súčasnej ľudskej spoločnosti. Veľmi často ich používame pre zábavu a relax vo voľnom čase a ešte častejšie v rámci pracovných povinností. Umožňujú nám pracovať efektívnejšie, pričom človek vynakladá oveľa menšie úsilie v porovnaní so situáciou, kedy by pracoval bez modernej techniky. Stále častejšie je na počítačové systémy prenášaná určitá zodpovednosť, napríklad v rámci automatizácie kritických riadiacich procesov (napr. riadenie časti letového systému lietadla, riadenie atómovej elektrárne, a i.). S týmto trendom vzrastajú aj požiadavky na komplexnosť a spoľahlivosť počítačových systémov.

V dobe, kedy je kvôli vysokej zložitosti systémov prakticky nemožné zaručiť ich spoľahlivosť tradičným testovaním a formálne techniky vyžadujú značné úsilie a skúsenosti, nastupuje na scénu funkčná verifikácia. Tá do určitej miery nahrádza techniku testovania prototypov, keď umožňuje overiť chovanie navrhovanej jednotky voči špecifikácii ešte pred začatím výrobného procesu. V súčasnosti je funkčná verifikácia neoddeliteľnou súčasťou návrhového cyklu hardvéru [5]. Podľa [1] je jej venovaných 60% až 80% úsilia v rámci procesu návrhu počítačového systému a výhodou je, že dokáže predchádzať značným finančným stratám spôsobených prehliadnutím chýb v návrhu, ktoré sú typicky po realizácii návrhu v kremíku neodstrániteľné alebo odstrániteľné len za vysokú cenu. Príkladom môže byť asi najznámejšia chyba v procesoroch Intel P5 Pentium z roku 1994, známa ako *Pentium FDIV bug* (FDIV označuje inštrukciu delenia v plávajúcej desatinnej čiarke). Chyba spôsobovala za určitých okolností chybu v delení a firmu Intel podľa jej prehlásení stála 475 miliónov amerických dolárov v spojení s výmenou chybných procesorov. Samozrejme, so vzrastajúcou komplexnosťou návrhu vzrastá aj zložitosť funkčnej verifikácie, preto je nutné vhodne voliť dostupné nástroje a postupy. Jedným z nich je aj použitie štandardizovaného jazyka SystemVerilog, ktorý bol špeciálne navrhnutý ako jazyk pre podporu funkčnej verifikácie a navyše poskytujúci aj prostriedky pre návrh hardvéru. V priebehu vývoja techník funkčnej verifikácie boli vytvorené viaceré metodiky, ktoré poskytujú podrobné návody ako pristupovať k procesu funkčnej verifikácie tak, aby bol jej výsledok čo najefektívnejší a cesta k nemu čo najjednoduchšia.

Ďalšou cestou v urýchľovaní vývoja komplexných počítačových systémov je využitie techniky súbežného návrhu hardvéru a pre neho dostupného softvéru. V prípade vývoja procesorov sa za týmto účelom využívajú jazyky pre popis architektúry procesoru a jeho inštrukčnej sady. Pomocou sady automatických nástrojov je možné z popisu procesora vytvoriť jeho model a vývojové prostriedky pre jeho programovanie, či písanie aplikácií, ktoré by na ňom mali bežať. Vďaka tomu je možné vytvoriť pre daný procesor aplikáciu ešte pred jeho realizáciou v hardvéri, pričom jej ladenie môže byť uskutočnené na simulovanom modeli procesora. Príkladom vývojového prostredia, ktoré pracuje na zmienenom princípe, je systém Codasip [6]. Za účelom overenia, že model takto navrhovaného procesoru zodpovedá špecifikácii, je možné uplatniť techniky funkčnej verifikácie. Cieľom tejto práce preto bude návrh metodiky, ako začleniť proces funkčnej verifikácie do vývojového cyklu automaticky generovaných procesorov v systéme Codasip.

2 Funkčná verifikácia

Hlavnou úlohou funkčnej verifikácie je ukázať, že hardwarový návrh implementuje požadovanú funkčnosť definovanú v jeho špecifikácii. Tento proces je vo všeobecnosti náročný a vyžaduje adekvátne množstvo času. Navyše každý návrhár či verifikátor môže pochopiť špecifikáciu rozdielne, pretože špecifikačné dokumenty sú do určitej miery otvorené pre interpretáciu. Nie vždy to však býva na škodu a rozdielny pohľad na danú problematiku môže vyústiť do odhalenia chyby v implementácii, ktorá by inak nemusela byť odhalená [11].

Funkčná verifikácia je založená na simulácii, no rozširuje ju o použitie sofistikovaných techník, pre dosiahnutie väčšej efektivity a redukciu času potrebného pre celý verifikačný proces. Niektoré z najdôležitejších princípov sú popísané v tejto kapitole.

2.1 Generovanie náhodných stimulov

Generovanie náhodných stimulov s obmedzujúcimi podmienkami (angl. *constrained-random stimulus generation*) je kľúčová technika pre preverovanie komplexných systémov. Umožňuje zautomatizovať vytváranie testovacích prípadov a generovanie veľkého množstva stimulov. Obmedzujúcimi podmienkami (angl. *constraints*) je možné zabezpečiť generovanie len povolených hodnôt, ktoré pre daný prípad majú zmysel a môžu sa v reálnej prevádzke vyskytnúť. Využitím tejto techiky je možné odhaliť neočakávané chybové stavy, ktoré na prvý pohľad nie sú zrejmé. Jedná sa napríklad o vygenerovanie vstupov, ktoré zachytia hraničné stavy jednotky, či vstupov, ktoré dostanú jednotku do stavu, o ktorom návrhár na základe špecifikácie vôbec nepredpokladal, že môže spôsobiť chybu. Pre zhodnotenie priebehu verifikácie, v ktorej je použitá technika generovania náhodných stimulov, sa odporúča použiť techniku pokrytia, o ktorej pojednáva nasledujúca podkapitola.

2.2 Pokrytie

Pokrytie (angl. *coverage*) je technika, ktorá nám poskytuje spätnú väzbu o priebehu verifikácie. Počas jednotlivých simulačných behov sú zaznamenávané určité metriky, ktoré sú na konci simulácie vyhodnotené a interpretované. Existujú dva typy pokrytia:

- **Pokrytie kódu** (angl. *code coverage*). Pokrytie kódu dokáže identifikovať časť zdrojového kódu modelu verifikovanej jednotky, ktorá bola resp. nebola v priebehu verifikácie vykonaná. Nízke pokrytie indikuje, že niektorá časť kódu modelu nebola prevedená, pričom nie je vylúčené, že práve táto časť môže obsahovať chyby. Naopak 100% pokrytie nezaručuje korektnosť, či kompletnosť verifikácie, napriek tomu je dosiahnutie čo najväčšieho pokrytia veľmi žiaduce.

  Existujú viaceré prístupy pre meranie pokrytia kódu, z ktorých najpopulárnejšie sú pokrytie riadkov (angl. *statement coverage* - sekvencia riadkov je označená ako pokrytá ak je jeden z nich vykonaný), pokrytie cesty (angl. *path coverage* - metrika sledujúca počet spôsobov vykonania sekvencie príkazov, napr. ktoré vetvy príkazu `if` boli realizované), pokrytie výrazov (angl. *expression coverage* - pokrytie všetkých možných hodnôt vo výrazoch, napr. `((a==b) || !c)`) a pokrytie FSM (angl. *FSM coverage*,

pokrytie prechodov medzi stavmi konečného automatu, *FSM* (angl. *finite state machine*), teda väčšinou príkazu `case`).

- **Funkčné pokrytie** (angl. *functional coverage*). Funkčné pokrytie zaznamenáva relevantné metriky (napr. dĺžku sieťového paketu, operačný kód inštrukcie, ...) a vďaka nim poskytuje možnosť určiť mieru kompletnosti verifikácie vzhľadom k špecifikácii, tzn. akú veľkú časť funkčnosti hardvéru sa nám podarilo preveriť. Cieľom je dosiahnuť 100% pokrytie, čo v praxi znamená otestovať všetky stavy, do ktorých sa verifikovaná jednotka môže dostať, zadaním relevantných kombinácií vstupných hodnôt. Pre použitie funkčného pokrytia je nutné ručne nadefinovať, ktoré hodnoty alebo stavy budú sledované.

2.3 Formálne tvrdenia

Formálne tvrdenia (angl. *assertions*) sú tvrdenia o vlastnostiach systému, ktoré vždy platia, a ich porušenie vedie k chybovému stavu. Pravdivosť formálneho tvrdenia je potrebné vyhodnocovať v priebehu simulačného času (typicky v každom cykle synchronizačného hodinového signálu) na rozdiel napr. od príkazu `if`, ktorého podmienka je vyhodnotená v čase, keď je vykonávaný samotný príkaz `if`. Preto je na zápis formálneho tvrdenia nutné použiť špeciálny jazyk, napr. SVA (angl. *SystemVerilog Assertions*), či PSL (angl. *Property Specification Language*). To nám umožňuje vyhodnocovať a overovať tvrdenia ako napr. „Žiadosť musí byť nasledovaná potvrdením alebo zamietnutím v priebehu najviac 10 hodinových cyklov po jej vystavení.“, čo je nad možnosti jedného obyčajného `if` príkazu.

Formálne tvrdenia sa využívajú v technike verifikácie založenej na tvrdeniach (angl. a*ssertion-based verification*). Tvrdenia zachytávajú špecifikáciu a chovanie systému vo formálnej podobe. Chovajú sa ako monitorovacie prvky a ak dôjde k porušeniu jedného z nich v priebehu simulácie, simulácia sa preruší a táto skutočnosť je okamžite oznámená v čase, keď sa problém vyskytol. V takomto prípade je hľadanie chýb a ich následná oprava efektívnejšie ako spätné vyhľadávanie príčiny chybnej činnosti vo výstupe simulácie.

2.4 Verifikačný plán

Pred samotným začatím verifikačných simulácií a testovania je potrebné stanoviť si ciele a plán verifikácie. Verifikačný plán je dokument (alebo skupina dokumentov) špecifikujúci proces verifikácie a infraštruktúru testovacích prostredí. Obsahuje čo najdetailnejší plán priebehu verifikácie, to akým spôsobom sa bude verifikovať (nástroje, verifikačné prístupy) a aké testovacie prostredia budú vytvorené. Takisto stanovuje stav, kedy bude verifikácia kompletná do takej miery, aby verifikovaná jednotka spĺňala požadovaný stupeň funkčnosti (napr. mierou dosiahnutého pokrytia).

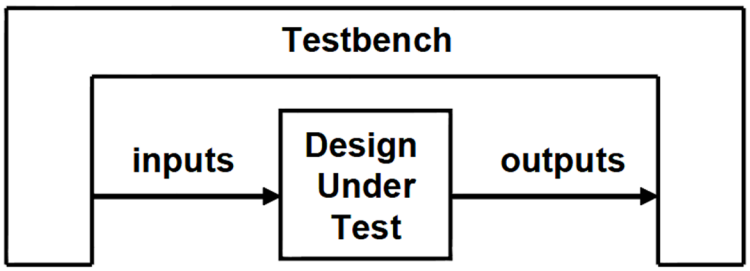
Verifikačný plán je založený na špecifikácii (špecifikačnom dokumente) navrhovaného systému. Špecifikačný dokument musí existovať pred zostavovaním verifikačného plánu a mal by čo najjednoznačnejšie popisovať funkčnosť systému. V priebehu verifikácie predstavuje akýsi zákon, podľa ktorého sa riešia rozdiely v očakávaných výstupoch a výstupoch produkovaných verifikovanou jednotkou.

Na základe verifikačného plánu je možné efektívne rozdeliť úlohy medzi jednotlivých členov verifikačného tímu, určiť požiadavky na zdroje a čas a naplánovať postup verifikácie. Samotný plán

potom slúži ako vodidlo či zaškrtávací zoznam určujúci oblasti a funkcie, ktoré už boli verifikované a tie, ktoré ešte len verifikované budú [2].

2.5 Verifikačné prostredie

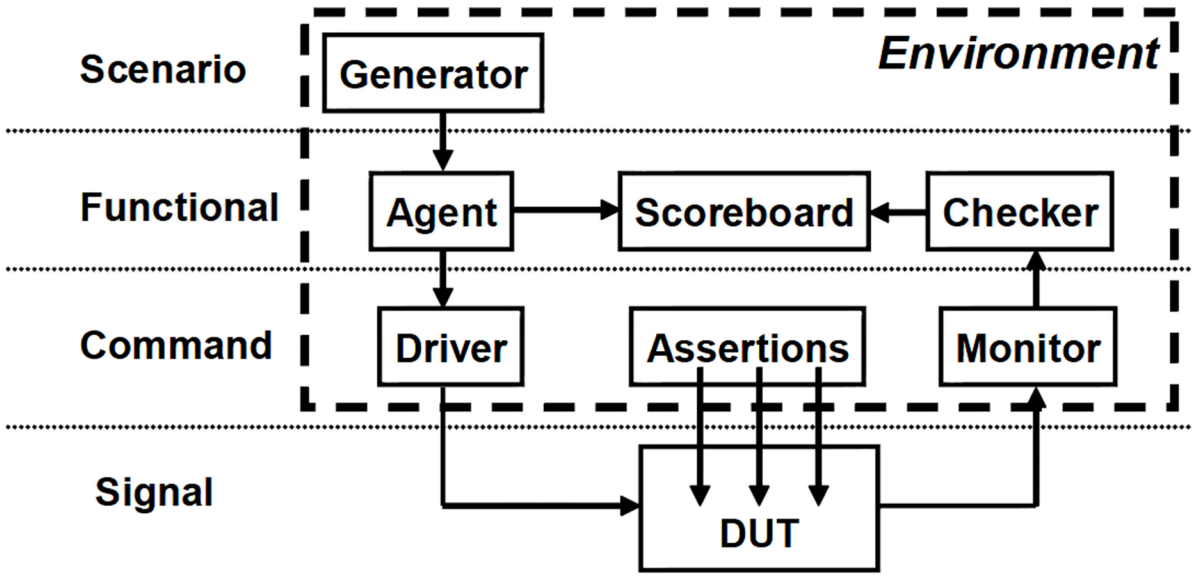
Verifikačné prostredie pre verifikovanú jednotku, DUT (angl. *Design Under Test*), je zobrazené na obrázku 2.1 (prevzatý z [3]). Jedná sa vlastne o testovacie prostredie zložené z modulov poskytujúcich signály pre DUT, dodržiavajúc protokol, ktorému DUT rozumie. Po spracovaní vstupu DUT vyprodukuje výstup, ktorý je zachytený a vyhodnotený verifikačným prostredím.



Obrázok 2.1 Jednoduchá schéma verifikačného prostredia.

2.5.1 Vrstvené testovacie prostredie

Testovacie prostredie zvyčajne pracuje na vyššej úrovni abstrakcie než na signálnej, preto každá moderná verifikačná metodika používa vrstvené testovacie prostredia (angl. *layered testbench*, viď obrázok 2.2 prevzatý z [3]). Pomocou nich je možné rozdeliť kód, ktorý implementuje testovacie prostredie, na samostatné časti. Tie sú zvyčajne vyvíjané nezávisle a každá definuje časť funkčnosti prostredia.



Obrázok 2.2 Architektúra vrstveného testovacieho prostredia.

Signálna vrstva (angl. *signal layer*) predstavuje najnižší stupeň abstrakcie. Zahŕňa verifikovanú jednotku spolu s jej vstupnými a výstupnými signálmi napojenými na testovacie prostredie.

Príkazová vrstva (angl. *command layer*) abstrahuje jednotlivé signály do jednoduchých príkazov (napr. čítanie/zápis do zbernice). *Driver* zabezpečuje interpretáciu jednotlivých príkazov na hodnoty signálov, ktoré sú potom posielané na vstup DUT. *Monitor* zaznamenáva hodnoty výstupných signálov DUT, ktoré zoskupuje do príkazov. Formálne tvrdenia (angl. *Assertions*) pracujú na rozmedzí signálnej a príkazovej vrstvy. Sledujú jednotlivé signály, ale v rámci kontextu celého príkazu.

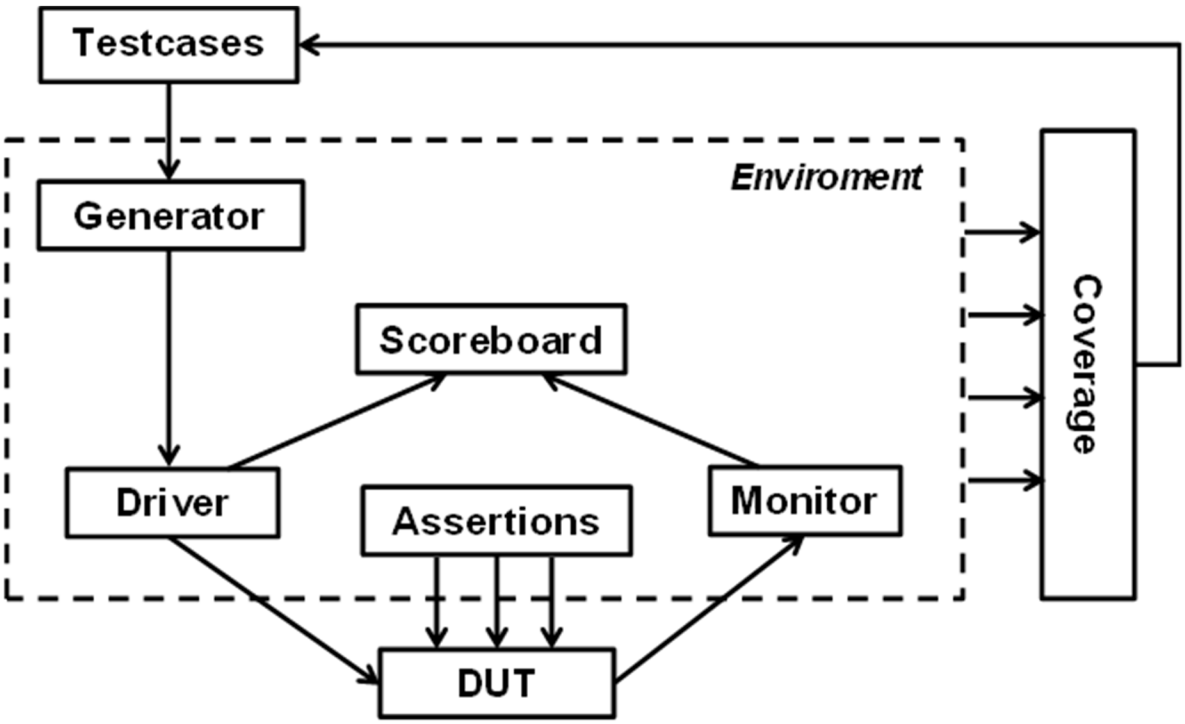
Funkčná vrstva (angl. *functional layer*) zabezpečuje rozčlenenie zložitejších príkazov (transakcií) na vyššom stupni abstrakcie (napr. príkaz na DMA prístup - priamy prístup do pamäte, angl. *Direct Memmory Access*) na jednoduchšie príkazy a naopak. *Agent* zasiela jednoduché príkazy *driveru* a zároveň posiela transakcie do *scoreboardu*, ktorý ich modifikuje podľa špecifikácie a vytvára tak očakávané výstupné transakcie. *Checker* transformuje príkazy získané z *monitora* do tvaru výstupných transakcií a porovnáva ich s predpovedanými transakciami v *scoreboarde*.

Scenárová vrstva (angl. *scenario layer*) obsahuje generátor (angl. *generator*), ktorý riadi funkčnú vrstvu generovaním rôznych prípadov použitia DUT. Jednotlivé prípady použitia môžu byť vytvorené priamo verifikátorom (angl. *directed tests*) alebo generované náhodne (angl. *constrained-random generation*).

Všetky bloky testovacieho prostredia sú napísané na začiatku verifikačného procesu. Počas verifikácie môžu byť vylepšované o novú funkčnosť, no nemali by sa meniť v závislosti na jednotlivých testovacích prípadoch.

2.5.2 Verifikácia riadená pokrytím a tvorba testov

U väčšiny súčasných verifikačných prostredí je možné rozoznať základné komponenty, ktoré sú uvedené na obrázku 2.3 (podľa [10]).



Obrázok 2.3 Verifikačné prostredie.

Agent a *checker* zdanlivo v obrázku chýbajú, no nie je tomu tak. Počet vrstiev verifikačného prostredia závisí na zložitosti DUT. U jednoduchších DUT nie je potrebné explicitne vyčleňovať jednotlivé vrstvy, a je možné funkčnosť jednoduchších komponentov združovať do komplexnejších celkov. V obrázku 2.3 je funkcia *agenta* obsiahnutá v *driveri*. Ten je teda schopný prijímať transakcie na vysokej úrovni abstrakcie vytvárané generátorom, prípadne do nich vložiť chyby či oneskorenia a rozčleniť ich na signály určené na vstup DUT. *Driver* takisto preposiela transakcie do *scoreboardu*, v ktorom je implementovaná porovnávacia funkčnosť *checkera*.

Definícia *scoreboardu* nie je štandardizovaná [1]. Niektoré výklady považujú *scoreboard* za dátovú štruktúru pre ukladanie predpovedaných hodnôt na základe vstupných transakcií, a samotné predpovedanie týchto hodnôt je realizované v inej časti testovacieho prostredia prostredníctvom transformačnej funkcie alebo referenčného modelu. *Scoreboard* ako je uvedený v obr. 2.3 je vnímaný ako komplexná samo-kontrolná (angl. *self-checking*) štruktúra, ktorá zahŕňa mechanizmus na predpoveď výstupu DUT ako aj štruktúru slúžiacu pre ukladanie tejto predpovede. Je v ňom teda implementovaná funkčnosť DUT nezávisle na jej implementácii vychádzajúc čisto zo špecifikačného dokumentu navrhovanej hardvérovej jednotky. Transakcie prijaté od *drivera* sú spracované tak, ako by ich mala spracovať aj DUT a sú uložené do transakčnej tabuľky. Po spracovaní transakcie v DUT je výstup prijatý *monitorom*, ktorý ho vo forme výstupnej transakcie prepošle do *scoreboardu*. Nasleduje porovnanie takto prijatej skutočnej výstupnej transakcie s očakávanou transakciou z transakčnej tabuľky. V prípade zhody je z nej transakcia odstránená. Ak po skončení simulácie transakčná tabuľka nie je prázdna, znamená to chybu v návrhu, tzn. niektoré výstupy sa nezhodovali, či jeden alebo viacero výstupov, ktoré mali byť vypočítané DUT, chýba. Existujú dva prístupy pri porovnávaní transakcií v *scoreboarde*:

- **FIFO prístup.** Predpovedané výstupy sú ukladané spôsobom FIFO (angl. *first-in-first-out*) a výstup DUT prijatý od monitora je porovnávaný len s prvým prvkom tabuľky. Tento prístup je vhodné použiť ak vieme, že DUT spracováva vstupy priebežne a poradie výstupných hodnôt bude zodpovedať poradiu adekvátnych vstupných hodnôt.

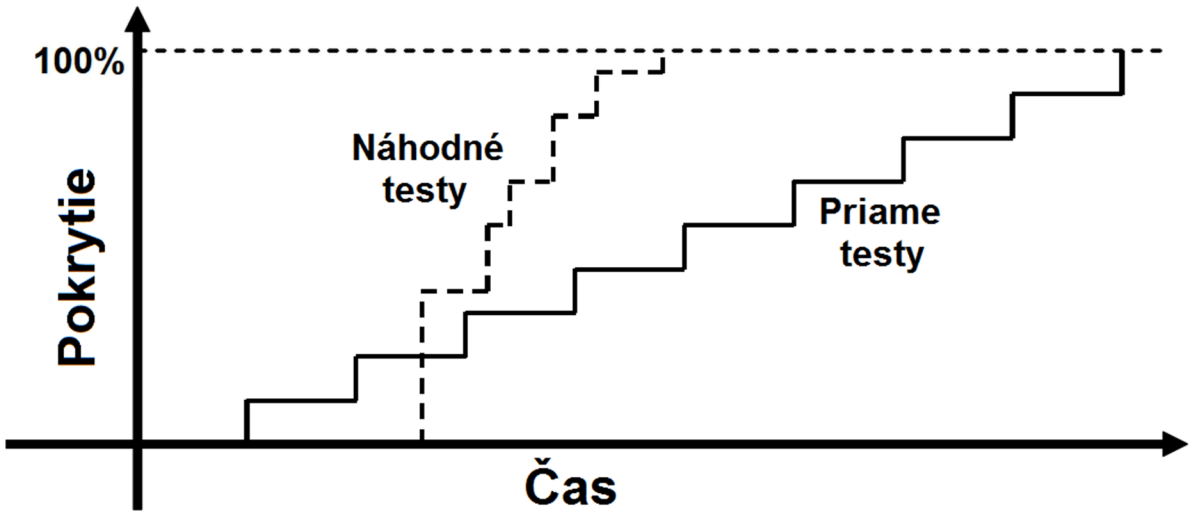
- **Porovnanie všetkých transakcií v tabuľke.** Porovnávanie sa neobmedzuje na prvý prvok tabuľky, ale porovnajú sa postupne všetky transakcie v nej, pokiaľ sa nenájde zhodná transakcia alebo kým sa neprehľadajú všetky záznamy. Tento prístup sa používa, ak DUT nezaručuje poradie spracovania vstupov napr. v procesore, kde pri použití zreťazenia (angl. *pipeline*), je možné, že výsledok operácie celočíselného sčítania bude na výstupe skôr, ako výsledok operácie delenia v plávajúcej desatinnej čiarke. Ďalším príkladom je strata dát v priebehu výpočtu (napr. pri testovaní jednotky implementujúcej chovanie sieťového zariadenia, ktoré môže niektoré vstupné pakety za určitých okolností zahodiť.

Výrazný podiel na úspešnosti a efektívnosti verifikácie má kvalita testov. Pre komplexné hardvérové návrhy je nutné pre preverenie všetkých možných stavov DUT vytvoriť veľké množstvo testov a vstupných stimulov. Rozlišujeme dva spôsoby tvorby testov a generovania stimulov:

- **Priame testy** (angl. *directed tests*). Sú vytvárané verifikačným inžinierom a slúžia na otestovanie konkrétnej funkčnosti. Stimuly sú vytvárané manuálne, aby cielene preverili požadovanú funkčnosť. Tento prístup je pri väčších projektoch nevýhodný pre svoju časovú náročnosť a fakt, že takto vytvorené testy väčšinou odhalia len predvídateľné chyby.

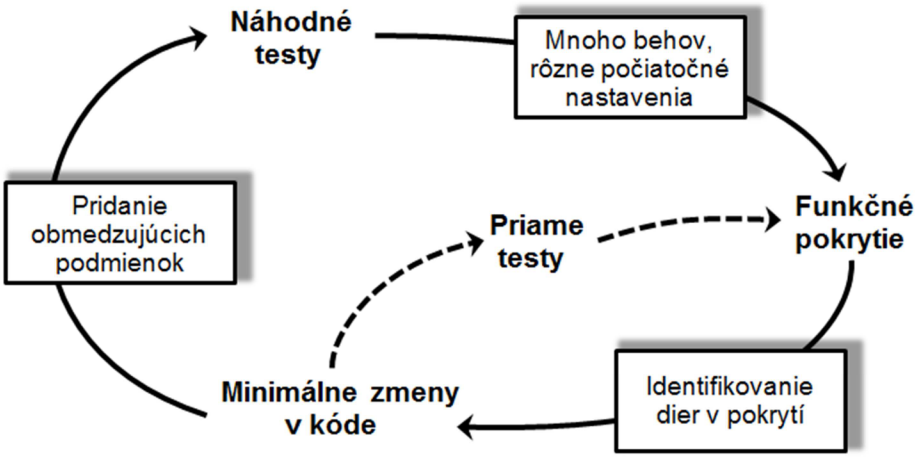
- **Testy s generovaním náhodných stimulov s obmedzujúcimi podmienkami** (angl. *constrained-random based stimulus generation tests*). Využívajú náhodné generovanie vstup-

ných stimulov, čím dokážu preveriť širokú oblasť stavového priestoru verifikovanej jednotky a odhaliť aj nepredpokladané chyby. Oproti priamym testom je počiatočné úsilie vynaložené na tvorbu náhodných testov väčšie, no obyčajne dokážu preveriť stavový priestor rýchlejšie ako priame testy a tak dosiahnuť lepšie pokrytie v kratšom čase (Obrázok 2.4 prevzatý z [3]).



Obrázok 2.4 Porovnanie spôsobov tvorby testov.

Zavedením spätnej väzby pomocou pokrytia v kombinácii s náhodným generovaním stimulov je možné dosiahnuť významné zefektívnenie verifikačného procesu. Táto technika sa nazýva verifikácia riadená pokrytím (angl. *coverage-driven random based verification*). Spočíva vo využívaní metrík, hlavne funkčného pokrytia, pri definovaní a pridávaní obmedzení v generátore transakcií. V rámci verifikačného prostredia sú definované body pokrytia (angl. *coverage points*), ktoré sledujú určité vybrané hodnoty alebo stavy a zisťujú, či počas verifikácie nadobudli všetky zaujímavé a relevantné hodnoty. Body pokrytia je nutné voliť rozumne tak, aby zbierané metriky boli naozaj relevantné a čo najpresnejšie určovali skutočnú mieru funkčného pokrytia. Pri nadefinovaní príliš veľkého množstva bodov pokrytia, ktoré budú navyše sledovať nerelevantné hodnoty, prípadne na nevhodných miestach, môže byť funkcia a účinnosť tejto techniky značne degradovaná. Obrázok 2.5 (prevzatý a upravený z [3]) ukazuje proces verifikácie riadenej pokrytím.



Obrázok 2.5 Verifikácia riadená pokrytím.

Pre preverenie funkčnosti, ktorú z nejakého dôvodu nie je možné preveriť pomocou náhodných testov, je možné napísať priame testy. Obrázok 2.3 ukazuje príklad verifikačného prostredia využívajúceho popisovaný prístup. Zbierané metriky poskytujú spätnú väzbu pri zostavovaní obmedzení pre jednotlivé testovacie prípady (angl. *testcases*). Testovací prípad je skupina testov zameraná na preverenie určitej funkčnosti.

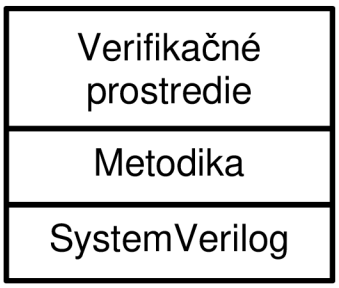
2.6 SystemVerilog

V súčasnosti je veľmi rozšíreným prostriedkom pre funkčnú verifikáciu jazyk SystemVerilog. Je to programovací jazyk pre popis, špecifikáciu a verifikáciu hardvéru, ktorý bol prijatý ako štandard (súčasná verzia IEEE Std 1800-2009 [12]). Ide o rozšírenie pôvodného štandardu Verilog (IEEE Std 1364-2005) najmä o funkčnosť potrebnú pre verifikáciu hardvéru. SystemVerilog umožňuje zvýšiť efektivitu verifikácie podporou množstva techník. Medzi najdôležitejšie techniky a prístupy podľa [8] patria:

- generovanie náhodných stimulov s obmedzujúcimi podmienkami,
- formálne tvrdenia,
- sledovanie pokrytia,
- definícia rozhrania (angl. *interface*) – mechanizmus umožňujúci zapuzdrenie vstupných a výstupných signálov verifikovaných jednotiek do jedného celku,
- volanie funkcií vytvorených v iných programovacích jazykoch pomocou DPI (angl. *Direct Programming Interface)* - rozhranie umožňujúce volať napr. C/C++/SystemC funkcie v rámci zdrojového kódu v SystemVerilogu a naopak,
- objektovo orientovaný prístup – uľahčenie programovania rozsiahlych systémov, ich udržovania a možnosť znovupoužiteľnosti komponent v iných systémoch.

2.7 Verifikačné metodiky

Verifikačné metodiky popisujú spôsob tvorby kooperujúcich verifikačných prostredí a ich komponent s ohľadom na znovupoužiteľnosť a efektivitu. Obrázok 2.6 ukazuje vzťah nasledujúcich metodík k SystemVerilogu a verifikačnému prostrediu.



Obrázok 2.6 Postavenie metodiky pri vývoji verifikačného prostredia.

- **Verification Methodology Manual (VMM)**

  VMM vznikla za spolupráce spoločností ARM a Synopsys. Bola predstavená v roku 2005 v knihe [4] (súčasná verzia VMM 1.2 bola vydaná v roku 2009). Definuje priemyselné postupy pre vytváranie robustných, znovupoužiteľných a rozšíriteľných verifikačných prostredí

v jazyku SystemVerilog. Obsahuje súhrn návodov a rád ako sa vyhnúť najčastejším chybám. Súčasťou je knižnica *WMM Standard Library* poskytujúce základné triedy pre tvorbu pokročilých testovacích prostredí, a *WMM Aplications* definujúca množiny zložitejších funkcií, ktoré riešia opakujúce sa spoločné požiadavky a problémy vo verifikačných prostrediach (napr. verifikácia registrov na čipe).

- **Open Verification Methodology (OVM)**

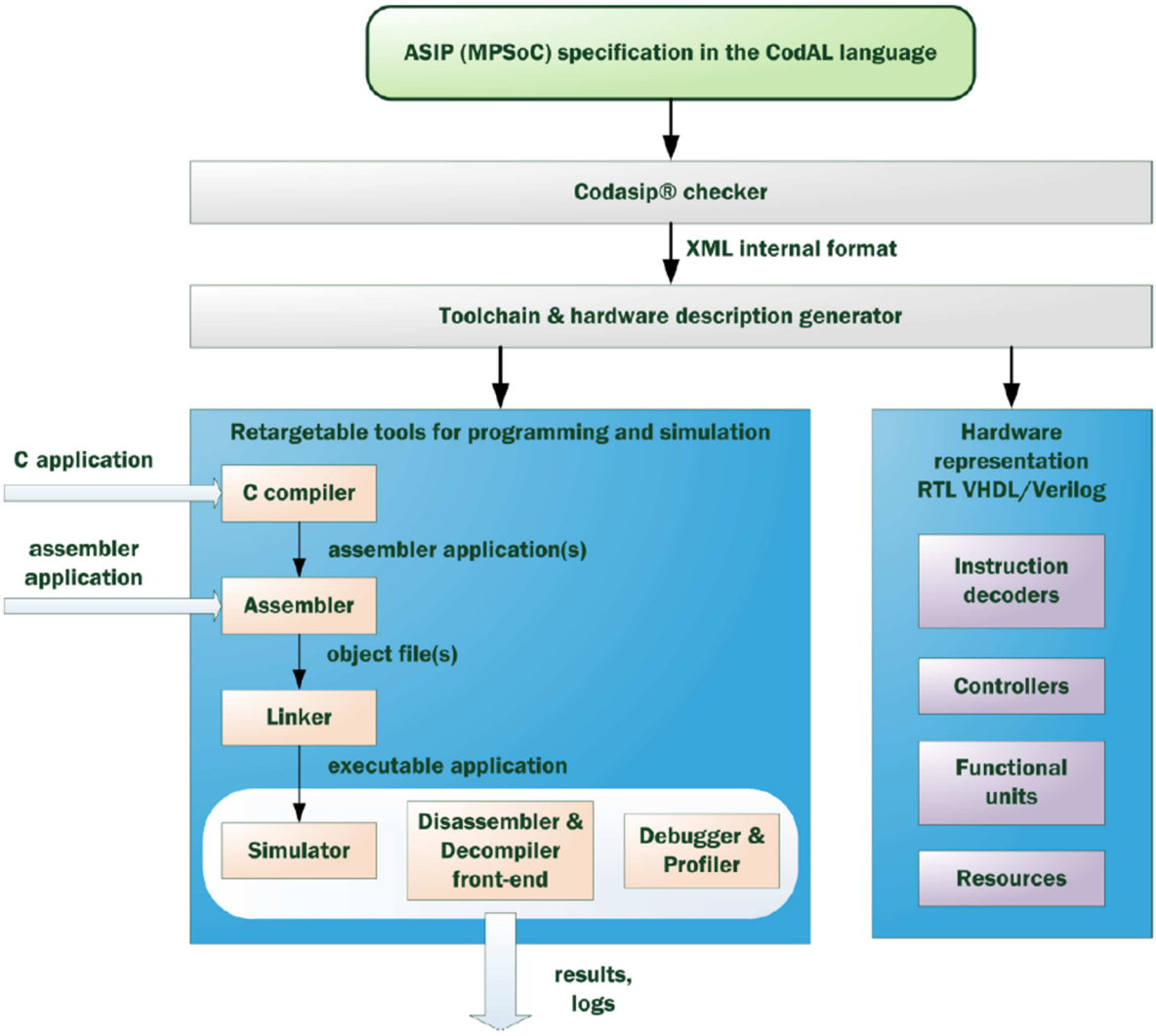
  Bola vytvorená na základe spolupráce firiem Mentor Graphics a Cadence. Verzia OVM 1.0 vyšla v roku 2008 a v súčasnosti je vo verzii OVM 2.1.2 (2011, po prihlásení dostupná z [13]). Je to multijazyčná metodika založená na verifikácii riadenej pokrytím. Obsahuje knižnicu tried, ktoré slúžia ako stavebné bloky pri vývoji znovupoužiteľných verifikačných komponentov a verifikačného prostredia, postupy, návody a príklady použitia.

- **Universal Verification Methodology (UVM)**

  UVM je odvodená od OVM. Vytvorila ju skupina expertov zastupujúcich viaceré spoločnosti pod záštitou spoločnosti Accellera. Vznikla tak štandardizovaná metodika UVM 1.0 (2011) založená na SystemVerilogu doplnená o funkčnosť VMM a OVM. Je spätne kompatibilná s OVM a podporovaná všetkými EDA spoločnosťami (angl. *Electronic Design Automation*). Oproti OVM je rozšírená napr. o mechanizmus monitorovania živosti verifikácie či mechanizmus uľahčujúci „upratovanie“ na konci simulácie [9]. Podporuje verifikáciu od blokovej úrovne až po systémovú, založenú na pokrytím riadenej verifikácii. Súčasná verzia UVM 1.1 je dostupná z [13].

3 Codasip

Vývojové prostredie Codasip je skupina nástrojov pre súbežný návrh hardvéru a softvéru. Uľahčuje tvorbu aplikačne špecifických inštrukčných procesorov ASIP (angl. *Application Specific Instruction-set Processor*) a viacprocesorových systémov na čipe MPSoC (angl. *Multiprocessor System on Chip*). Využíva jazyk pre popis architektúry CodAL, pomocou ktorého je možné namodelovať jednotlivé ASIP a MPSoC. Na základe modelu je Codasip systém schopný pomocou generátorov vytvoriť nástroje umožňujúce programovanie a simuláciu cieľového systému, a takisto syntetizovateľný hardvérový popis pre realizáciu navrhnutého modelu v programovateľnom poli logických hradiel FPGA (angl. *Field-Programmable Gate Array*), či ASIP. Obrázok 3.1 prevzatý z [6] demonštruje štruktúru systému Codasip.



Obrázok 3.1 Systém Codasip.

V priebehu kompilácie programu napísaného v jazyku CodAL slúži *Codasip checker* na detekciu hrubých chýb v návrhu architektúry (napr. viacnásobný prístup k zdrojom, ktoré viacnásobný prístup nepodporujú). Po skompilovaní modelu architektúry je možné vygenerovať nasledujúce nástroje:

- **Assembler**. Slúži na preklad strojového kódu do binárnych objektových súborov.
- **Disassembler**. Dokáže spätne previesť objektový súbor do pôvodného strojového kódu, ktorý môže byť znova preložený assemblerom.
- **Linker**. Spája viaceré objektové súbory do spustiteľného programu a rieši premiestňovanie adries (cieľov skokov, dát) v rámci pamäti.
- **C kompilátor**. Umožňuje preložiť program napísaný v jazyku C do špecifického strojového kódu.
- **C dekompilátor**. Reverzná činnosť oproti kompilátoru.
- **Simulátory**. Codasip podporuje generovanie niekoľkých simulátorov buď interpretovaných alebo kompilovaných na rôznych úrovniach detailnosti a presnosti simulácie, tzn. na inštrukčnej úrovni (angl. *instruction accurate*), na úrovni hodinových cyklov (angl. *cycle accurate*) a na úrovni RTL (angl. *Register Transfer Level*).
- **Profiler**. Nástroj pre zbieranie štatistík o priebehu simulácie, na základe ktorých je návrhár schopný určiť najdrahšie operácie a optimalizovať buď program pre navrhnutý systém alebo systém samotný.
- **Debugger**. Prostredie pre hľadanie a opravu chýb počas návrhu.

Pomocou týchto nástrojov je možné vytvárať a optimalizovať softvér pre navrhovaný hardvérový systém i samotný systém.

Súčasťou systému Codasip je generátor, schopný z popisu architektúry v jazyku CodAL vygenerovať popis cieľového systému v jazyku VHDL (angl. *Very High-speed Integrated Curcuit Hardware Description Language*). Tento syntetizovateľný kód je rozdelený do niekoľkých zdrojových súborov VHDL popisujúcich jednotlivé funkčné jednotky, registre, pamäte, radiče či dekódery. Takisto je vygenerovaný VHDL súbor používajúci vyššie uvedené komponenty, ktorý predstavuje navrhovaný systém ako celok.

3.1 Codasip IDE

Vývojové prostredie, IDE (angl. *Integrated Development Enviroment*), pre systém Codasip pozostáva z troch vrstiev: prezentačnej, strednej (angl. *middleware*) a simulačnej. Komunikácia medzi nimi je realizovaná pomocou protokolu TCP/IP, čo umožňuje prevádzkovať každú časť vývojového prostredia na rôznych koncových staniciach v rámci siete.

Prezentačná vrstva slúži ako „*front-end*“ pre vývojára. Prijíma jeho príkazy a zobrazuje dôležité informácie. Codasip IDE ponúka dve formy prezentačnej vrstvy. Prvá, *Codasip Studio*, poskytuje grafické užívateľské rozhranie založené na platforme Eclipse. Druhá, *Codasip Command Line* poskytuje rozhranie príkazového riadka umožňujúc použiť pokročilé techniky, napr. skriptovanie.

Stredná vrstva spracováva príkazy prijaté od prezentačnej vrstvy. Ak je schopná prijatý príkaz vykonať (napr. vygenerovanie niektorého dostupného nástroja), prevedie ho a výsledok oznámi prezentačnej vrstve. Príkazy, ktoré vykonať nedokáže, preposiela simulačnej vrstve.

Simulačná vrstva sa skladá zo simulátorov jednotlivých procesorov. Pri simulovaní viacprocesorového systému má každý procesor vlastný simulátor, ktorý môže byť umiestnený na ľubovoľnej koncovej stanici v sieti. O umiestňovanie simulátorov na konkrétne stroje sa stará stredná vrstva.

3.2 Jazyk CodAL

Jazyk CodAL spadá do skupiny jazykov ADL (angl. *Architecture Description Language*). Jedná sa o zmiešaný typ ADL, ktorý popisuje jednak architektúru procesorov a SoC (angl. *System on Chip*) na vyššom stupni abstrakcie a zároveň ich inštrukčnú sadu. Podporuje modelovanie viacprocesorových systémov a súbežný návrh softvéru a hardvéru. Informácie obsiahnuté v zdrojovom kóde v jazyku CodAL je možné podľa [7] rozdeliť do štyroch kategórií:

- popis inštrukčnej sady modelu,
- popis časovania architektúry, tzn. aktivácia jednotlivých stavebných blokov procesora,
- popis chovania stavebných blokov procesora,
- popis štruktúry architektúry, tzn. prepojenia jednotlivých stavebných blokov procesora.

Popis procesoru v jazyku CodAL vyžaduje základnú štruktúru zdrojového súboru nezávisle od zložitosti, či detailnosti modelu. V rámci nej je potrebné definovať dve hlavné časti popisu. Prvá z nich, popis zdrojov, obsahuje popis hardvérových prvkov procesoru. Zahŕňa napr. špecifikáciu registrov, pamätí, zberníc, ich vzájomného mapovania, programového čítača, či iných prvkov. Druhá časť obsahuje popis inštrukčnej sady procesora a popis reakcií na určité udalosti (napr. reakcia na inštrukciu). Minimálne musí obsahovať popis udalostí `reset` (čo sa stane pri resetovaní procesora), `halt` (čo sa má uskutočniť, ak bude procesor zastavený) a `main` (úkony, ktoré sa majú vykonať v rámci každého hodinového cyklu procesora).

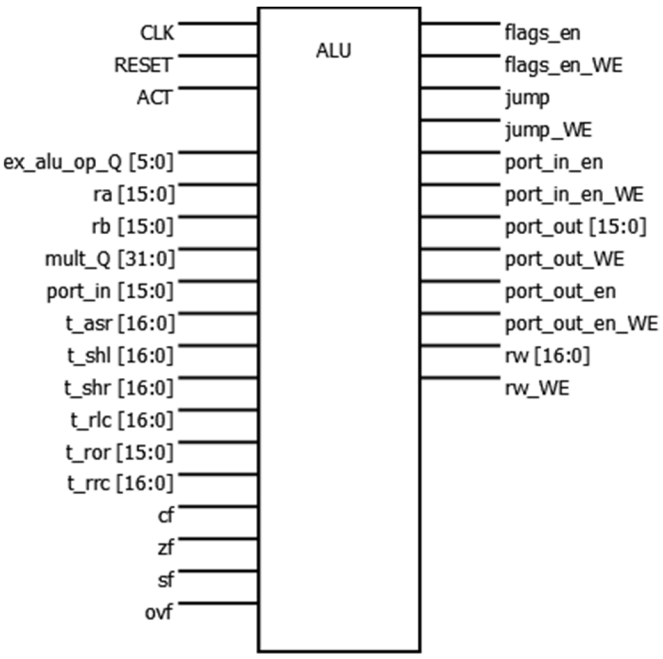
4 ALU procesora ADOP

Aritmeticko logická jednotka, ALU (angl. *Arithmetic Logic Unit*), je základný stavebný blok každého CPU (angl. *Central Processing Unit)*. ALU procesora ADOP je číslicový, čisto kombinačný obvod. Zabezpečuje vykonávanie základných aritmeticko-logických operácií s číslami v pevnej rádovej čiarke ako sú aritmetické alebo logické posuny a rotácie, celočíselné aritmetické operácie a logické operácie. V nasledujúcom texte je popísané rozhranie ALU procesora ADOP a operácie ňou podporované sú uvedené v prílohe 3.

4.1 Rozhranie ALU

Hardvérový model ALU, ktorej verifikácia je cieľom tejto práce, je vygenerovaný z popisu v jazyku CodAL. V priebehu práce sa model ALU kvôli optimalizáciám mierne modifikoval, a preto sa verifikovali celkovo dve verzie ALU, ktoré sa líšili rozhraním. Rozhranie pre starší model je na obrázku 4.1 a je rozobrané ďalej v texte. Jediná zmena rozhrania nového oproti staršiemu modelu spočívala v chýbajúcich výstupných signáloch príponou `_WE`. Ak je pri popisoch v ďalšom texte podstatný rozdiel medzi starším a novším modelom, vždy je explicitne uvedené, o ktorý model sa jedná. Ak nie, popis platí pre oba.

Vstupné rozhranie ALU obsahuje štandardné vstupy pre hodinový a resetovací signál (`CLK`, `RESET`), ktorých význam je všeobecne známy. Význam ostatných vstupných a výstupných signálov je popísaný ďalej v tejto podkapitole.



Obrázok 4.1 Schéma rozhrania ALU ADOP.

- `ACT` – Signál `ACT` slúži na aktiváciu jednotky. Ak je signál v logickej „1“, jednotka pracuje a produkuje výstupy podľa nastavenej operácie. V logickej „0“ by mala byt jednotka zastavená a neprevádzať žiaden výpočet.

- `ex_alu_op_Q` – Určuje operáciu, ktorú má ALU vykonať. Zoznam a jednoduchý popis jednotlivých operácií je uvedený v prílohe 3.
- `ra`, `rb` – Predstavujú 16 bitové dátové vstupy, pre prvý a druhý vstupný operand ALU.
- `mult_Q` – Signál slúži pre vstup pomocných dát, aby po integrovaní ALU do procesora ADOP bolo možné realizovať násobenie.
- `port_in` – Pomocou operácie *IN* je možné dáta z tohto vstupu previesť na výstupný port `rw`.
- `t_asr`, `t_shl`, `t_shr`, `t_rlc`, `t_ror`, `t_rrc` – Predstavujú dátové vstupy pre operácie aritmetických alebo logických posunov a rotácií, v uvedenom poradí ide o aritmetický posun doprava, logický posun doľava, logický posun doprava, rotácia doľava cez *carry bit*, rotácia doprava a rotácia doprava cez *carry bit*.
- `cf`, `zf`, `sf`, `ovf` – Sú vstupy pre hodnoty príznakov z príznakového registra, v uvedenom poradí *carry flag*, *zero flag*, *sign flag* a *overflow flag*.

Výstupné rozhranie staršieho modelu ALU obsahuje ku každému signálu pre výstupné dáta signál s príponou `_WE` (angl. *Write Enable*), ktorý indikuje, že dáta na danom výstupe sú platné. Význam ostatných signálov spoločných pre oba modely je:

- `flags_en` – Určuje či vykonaná operácia mení príznaky.
- `jump` – Pri skokových operáciách udáva, či má alebo nemá byť prevedený skok.
- `port_in_en` – Výstup v logickej „1“ indikuje očakávané dáta na vstupe `port_in`.
- `port_out` – Pri operácii *OUT* je vstup z prvého operandu prepísaný na tento výstup.
- `port_out_en` – Určuje či sú na výstupe *port_out* pripravené dáta.
- `rw` – Slúži ako dátový výstup pre výsledok všetkých operácií okrem *IN* a *OUT*.

5 Návrh a implementácia verifikačného prostredia

Po analýze rozhrania a chovania sa verifikovanej jednotky bolo možné pristúpiť k tvorbe prostredia pre jej verifikáciu. Verifikačné prostredie pozostáva z dvoch hlavných častí. Prvá je zložená z tried základných komponent verifikačného prostredia navrhnutých čo najvšeobecnejšie tak, aby boli znovu použiteľné a bolo ich možné uplatniť bez zmeny pri verifikácii širokého spektra hardvérových jednotiek.

Druhá časť rozširuje triedy základných komponentov verifikačného prostredia o špecifické rysy dané verifikovanou jednotkou. Táto časť musí byť „ušitá na mieru“ každej jednotke, ktorú chceme verifikovať. Sem patrí okrem iného mapovanie rozhrania verifikovanej jednotky na rozhranie verifikačného prostredia, určenie pravidiel pre posielanie a odchytávanie vstupov a výstupov či už pre potreby samotnej simulácie, alebo sledovania funkčného pokrytia. V nasledujúcich podkapitolách sú detailnejšie opísané obe časti zahrňujúc hlavné implementačné detaily a vysvetlenie princípov.

5.1 Základ verifikačného prostredia

Verifikačné prostredie je založené na metodike VMM a OOP a jeho základ je uložený v zložke *ver_base* (viď príloha 2). Tá obsahuje súbory so základnými rodičovskými (bázovými) triedami potrebnými pri verifikácii, ktoré sú pri vytváraní verifikačného prostredia pre konkrétnu jednotku rozšírené princípom dedičnosti o špecifické funkcie a vlastnosti. Väčšina metód uvedených v týchto triedach je virtuálnych a je potrebné ich implementovať v triedach potomkov pri vytváraní konkrétneho prostredia. Ako príklad nevirtuálnych metód uvediem metódy *setEnabled* a *setDisabled* prítomné v bázových triedach *generátora*, *drivera* a *monitora*. Už ich názov napovedá, že sa jedná o metódy, ktoré uvedú jednotlivé jednotky do činnosti, respektíve ich vykonávanie zastavia a nie je ich potrebné implementovať v každej triede potomka, keďže ich funkčnosť sa nemení.

Trieda generátora transakcií je implementovaná v súbore *generator.sv*. Pri inštanciácii objektu generátora je vytvorené úložisko generovaných transakcií, tzv. `mailBox`. Činnosť generátora je implementovaná vo virtuálnej metóde `run`, a pozostáva z vytvorenia novej transakcie, vygenerovania náhodných hodnôt dátových zložiek transakcie a následného uloženia hotovej transakcie do `mailBoxu`. Formát generovanej transakcie je definovaný tzv. vzorom transakcie, ktorý je reprezentovaný premennou `blueprint` v objekte generátora. Bázová trieda pre transakciu je definovaná v súbore *transaction.sv*. Definuje virtuálne metódy pre kopírovanie transakcie `copy`, zobrazenie obsahu transakcie `display` a porovnanie obsahu transakcie s inou transakciou `compare`. Práve pomocou metódy `copy` sú v generátore z premennej `blueprint` vytvárané nové transakcie. Samotné generovanie náhodných hodnôt dátových zložiek transakcie je zabezpečené systémovou funkciou SystemVerilogu `$randomize`.

`MailBox` generátora slúži zároveň ako vstupný prúd transakcií pre *driver*. Jeho bázová trieda je implementovaná v súbore *driver.sv*. Princíp činnosti *drivera* spočíva vo vyberaní transakcií z `mailBoxu`, ktoré následne distribuuje inštancii *scoreboardu* a verifikovanej jednotke. Zasielanie transakcií jednotke sa deje pomocou rozhraní, ktoré sú opísané v ďalšej podkapitole. Pre odosielanie transakcií do *scoreboardu* sa využívajú tzv. *callback*-y.

Callback je trieda implementujúca dve virtuálne funkcie `pre_tr` a `post_tr`, ktoré vykonajú určitú operáciu nad transakciami, ktoré sú im predané ako parameter. Typicky sa jedná o modifikáciu transakcie podľa špecifikácie. Vo verifikačnom prostredí existujú dve triedy pre *callback*, vstupný a výstupný, implementované v súboroch *input_cbs.sv* a *output_cbs.sv*. Tieto triedy slúžia ako základ pre triedu *scoreboardu*, ktorá definuje ich virtuálne metódy tak, že pri ich zavolaní je transakcia spracovaná a pridaná do porovnávacej tabuľky *scoreboardu*, alebo je z nej odstránená a porovnaná so skutočným výstupom verifikovanej jednotky. Trieda porovnávacej tabuľky *scoreboardu* je definovaná v súbore *transaction_table.sv* a obsahuje základné metódy pre pridávanie a odoberanie transakcií.

Posledným základným prvkom je trieda *monitora* v súbore *monitor.sv*. Podobne ako *driver*, aj *monitor* komunikuje so *scoreboardom* pomocou *callback*-u a pre odchytávanie výsledkov produkovaných verifikovanou jednotkou sa využíva rozhranie. Základ verifikačného prostredia ďalej obsahuje súbory balíčkov pomocných matematických funkcií *math_pkg.sv* a balíček pre import rodičovských tried základných komponentov popísaných v tejto podkapitole sv*_common_pkg.sv*.

5.2 Verifikačné prostredie ALU

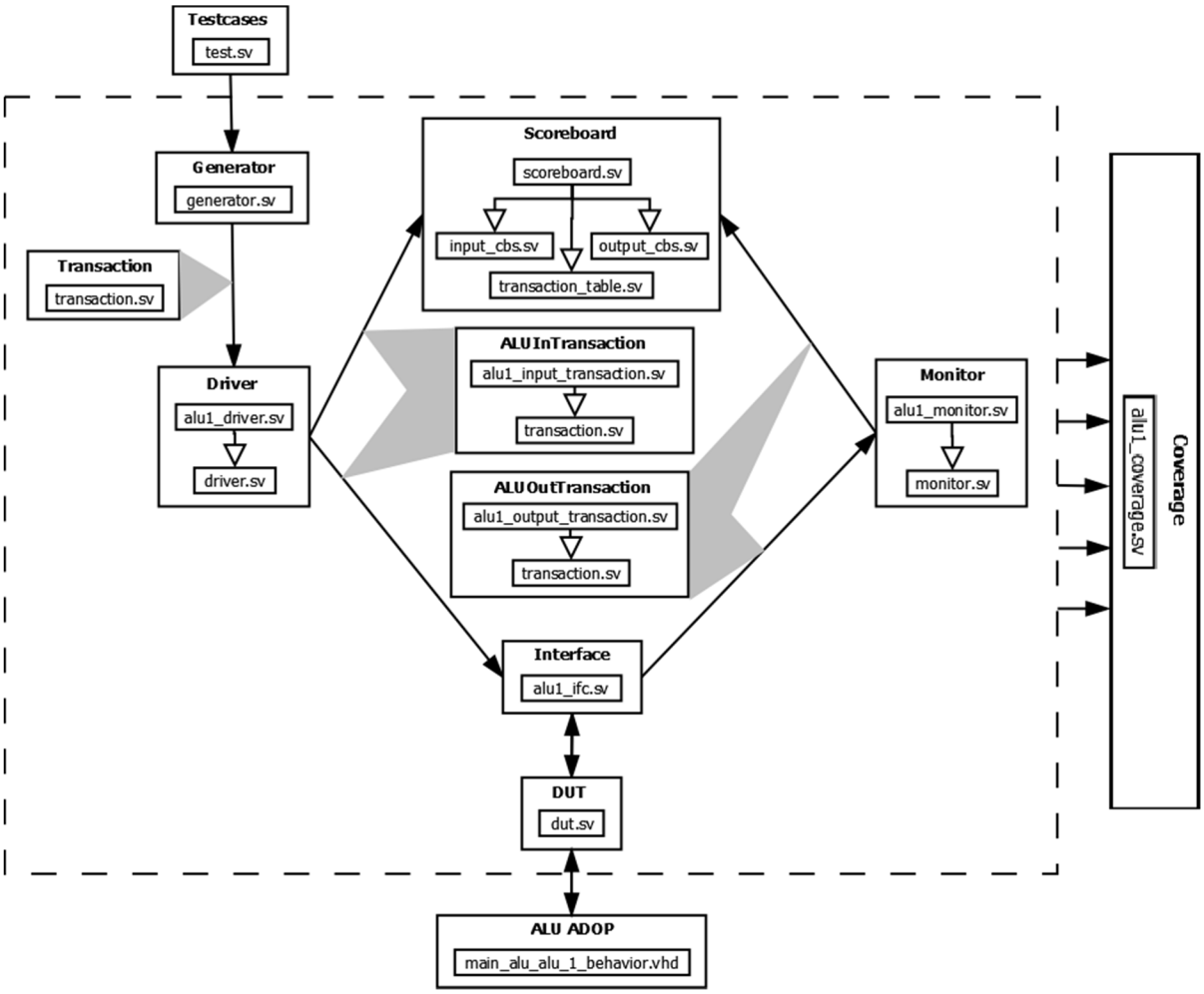
Verifikačné prostredie pre ALU tvoria súbory s triedami, ktoré sú vystavané na bázových triedach a prispôsobené verifikovanej jednotke (viď príloha 2). Obrázok 5.1 je analógiou k obrázku 2.3 a predstavuje verifikačné prostredie pre ALU procesora ADOP. Na obrázku nie sú znázornené súbory *sv_alu1_pkg.sv*, *test_pkg.sv*, a *testbench.sv*, patriace do prostredia. Prvý balíček slúži na zavedenie potrebných symbolov a deklarácií funkcií do iných súborov prostredia. Druhý obsahuje konštanty, parametre a generické parametre, pomocou ktorých je možné ovplyvniť priebeh verifikácie. Súbor *testbench.sv* je kostra verifikačného prostredia a špecifikuje prepojenie prostredia s verifikovanou jednotkou.

Testcases. Súbor *test.sv* obsahuje základný program, ktorý inštanciuje a inicializuje komponenty verifikačného prostredia a realizuje samotný verifikačný proces. Sú v ňom uložené taktiež jednotlivé testovacie prípady, či už v podobe priamych testov, alebo generátorom vytvorených automatických testov.

ALUInTransaction, ALUOutTransaction. Formát vstupných a výstupných transakcií je definovaný v súboroch *alu1_input_transaction.sv* a *alu1_output_transaction.sv*. Ich dátové zložky zodpovedajú signálom vstupného a výstupného rozhrania verifikovanej jednotky, ktoré sú v rámci jej verifikácie relevantné. V prípade tejto konkrétnej jednotky sú to všetky signály vstupného rozhrania, no pri výstupnom rozhraní sú to iba dátové výstupy. Všetky povoľovacie výstupy (signály s príponou `_WE`) je možné vynechať, pretože pri porovnávaní výstupov verifikovanej jednotky a výpočtu v *scoreboarde* nie sú potrebné. Ďalej sú v nich implementované virtuálne metódy z bázovej triedy *transaction.sv*. Obe implementujú metódu pre zobrazenie dátových zložiek, `display`. V triede výstupnej transakcie je navyše implementovaná metóda `compare` pre porovnanie transakcie s inou výstupnou transakciou, používaná v priebehu práce *scoreboardu*. Naopak v triede vstupnej transakcie je implementovaná metóda pre kopírovanie transakcie, `copy`. Oproti výstupnej transakcii sú v nej takisto definované obmedzenia platné pri generovaní hodnôt jej dátových zložiek, ktorých detailnejší popis je uvedený v podkapitole 5.4.3.

Interface. Vstupné a výstupné signály verifikovanej jednotky sú zapuzdrené do rozhrania v súbore *alu1_ifc.sv*. V ňom sú pomocou tzv. *modportov* identifikované ako vstupné, respektíve výstupné voči verifikačnému prostrediu a verifikovanej jednotke. Pre synchrónny prístup k jednotlivým signálom sú využité tzv. *clocking blocky*, ktoré umožňujú definovať okamih, kedy sa budú hodnoty daných signálov vzorkovať. V prípade vstupného rozhrania sú hodnoty poslané na vstup tesne pred

nasledujúcou nástupnou hranou, naopak výstupné signály sú vzorkované tesne po nástupnej hrane hodinového signálu.



Obrázok 5.1 Kompletné verifikačné prostredie.

DUT. Súbor *dut.sv* obsahuje modul, ktorý prostredníctvom definovaného rozhrania spája verifikačné prostredie s verifikovanou jednotkou. Mapuje rozhranie modelu ALU v jazyku VHDL na rozhranie implementované v SystemVerilogu, na základe čoho je potom možné poslať stimuly z prostredia do jednotky a prijať jednotkou produkované výstupy.

Scoreboard. Používaný *scoreboard* zahŕňa jednak dátovú štruktúru pre uloženie výsledkov, takzvanú transakčnú tabuľku (implementovanú v *transaction_table.sv*), ako aj samotný algoritmus pre výpočet hodnôt výstupných transakcií z prijatých vstupných transakcií z *drivera*. Tento výpočet spúšťa *driver* zavolaním virtuálnej metódy vstupného *callback*-u, `post_tr`. Metóda realizuje výpočet výstupných hodnôt na základe vstupnej transakcie a uloží ich v podobe výstupnej transakcie do transakčnej tabuľky. Odstraňovanie transakcií z tabuľky prebieha prostredníctvom metódy výstupného *callback*-u `post_tr`, ktorú volá *monitor* pri zbere výstupov z verifikovanej jednotky. Pri odstraňovaní je uplatnený princíp prvej zhody, čo znamená, že transakcia sformovaná *monitorom*, obsahujúca výstupy verifikovanej jednotky je porovnaná s prvou transakciou v transakčnej tabuľke (metódou `compare`). Nezhoda pri porovnaní znamená odchýlku vo výpočte verifikovanej jednotky od výpočtu v *scoreboarde*, čo vyvolá chybové hlásenie a pozastavenie verifikácie z dôvodu možnej prítomnosti chyby na jednej alebo druhej strane.

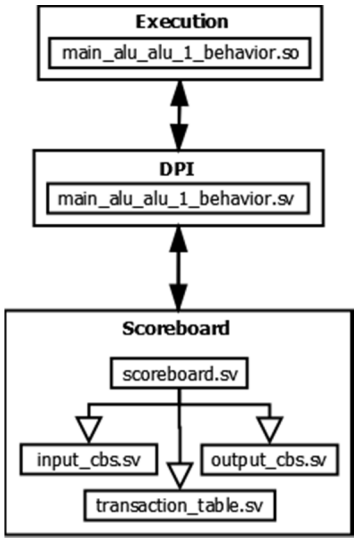
Driver. Činnosť *drivera* je implementovaná v súbore *alu1_driver.sv*. Spočíva vo vyberaní transakcií z `mailBox`-u, ktoré následne pošle *scoreboardu* (vstupný *callback*) a verifikovanej jednotke. V druhom prípade sa využíva definované vstupné rozhranie, konkrétne `iAluIn`, prostredníctvom ktorého sa v priebehu činnosti *drivera* propagujú vygenerované hodnoty vstupu, uložené v dátových zložkách transakcie, na vstupné signály DUT.

Monitor. Je definovaný v súbore *alu1_monitor.sv*. Odchytávanie výstupov jednotky je realizované cez výstupné rozhranie `iAluOut`, a je rozdielne pre starší a novší model. Vzorkovanie výstupov staršieho modelu sa riadi hodnotami jeho povoľovacích výstupných signálov (signály s príponou `_WE`). Keďže tieto signály nie sú v novšom modeli prítomné, prebieha zber výstupov novšieho modelu v každom takte s každou nástupnou hranou hodinového signálu. Zo zozbieraných výstupov sa sformuje výstupná transakcia, ktorá je pomocou výstupného *callback*-u zaslaná na spracovanie do *scoreboardu*.

Coverage. Trieda implementujúca zber štatistík funkčného pokrytia je implementovaná v súbore *alu1_coverage.sv*. Sú v nej deklarované body pokrytia, ktoré sa majú sledovať počas verifikačného behu. Trieda sleduje signály vstupného rozhrania, pri ich zmene si odchytí ich hodnoty a pridá ich do štatistík funkčného pokrytia. Detailnejší popis funkčného pokrytia je uvedený v podkapitole 5.4.

5.3 Výpočet v scoreboarde a DPI

Výpočet očakávaných výstupov v *scoreboarde* je možné realizovať dvoma spôsobmi. Prvý z nich implementuje výpočet priamo v *scoreboarde*, v tele funkcie `post_tr`. Druhý spôsob, použitý pri verifikácii novšieho modelu, využíva DPI rozhranie SystemVerilogu k volaniu sady externých funkcií implementovaných v jazyku C++. Tieto funkcie boli navrhnuté tak, aby kopírovali chovanie verifikovanej jednotky, takže pre predané vstupné hodnoty vypočítajú očakávané výstupy. Sú generované z popisu obvodu na vyššej úrovni abstrakcie v jazyku CodAL a sú považované za určitú formu špecifikácie. Nezhoda medzi výstupmi DUT a výstupmi sformovanými na základe výpočtu cez DPI je považovaná za chybu vo verifikovanej jednotke. Realizácia zapojenia výpočtu cez DPI je znázornená na obrázku 5.2.



Obrázok 5.2 Zapojenie výpočtu cez DPI.

Presunutie výpočtu očakávaných výstupov mimo prostredie SystemVerilogu je výhodné, hlavne ak je tento výpočet značne zložitý a tým pádom je verifikácia danej jednotky časovo náročná. C++ predstavuje vyhovujúce prostredie pre rýchle riešenie algoritmicky zložitých operácií, ktoré by potenciálne mohlo urýchliť priebeh verifikácie. Na druhej strane, pre funkčnosťou jednoduché jednotky by mohla už samotná réžia pri volaní takto importovaných funkcií spôsobiť určité spomalenie verifikácie. Porovnanie rýchlosti oboch prístupov je uvedené v podkapitole 6.4.

V priebehu kompilácie verifikačného prostredia je zo súboru *main_alu_alu_1_behavior.cpp* automaticky vytvorená knižnica *main_alu_alu_1_behavior.so*. Súbor *main_alu_alu_1_behavior.cpp* obsahuje implementáciu funkcií pre nastavenie hodnôt vstupných signálov a výpočet očakávaných výstupov, ktoré reprezentujú referenčný model pre verifikovanú jednotku. Spolu so súborom *main_alu_alu_1_behavior.sv* sú automaticky generované nástrojmi systému Codasip. Súbor *main_alu_alu_1_behavior.sv* je balíček deklarácií funkcií z knižnice *main_alu_alu_1_behavior.so*, ktoré majú byť cez DPI rozhranie importované do prostredia SystemVerilogu. Aby boli dané funkcie v tomto prostredí použiteľné, sú ich deklarácie upravené tak, aby spĺňali podmienky pre import cez DPI rozhranie a dodržiavali mapovanie dátových typov C++ na dátové typy SystemVerilogu a naopak podľa definície DPI. Pre ich následné použitie vo verifikačnom prostredí je súbor *main_alu_alu_1_behavior.sv* importovaný do súboru *scoreboard.sv*. Volanie importovaných funkcií vo funkcii `post_tr` sa potom z pohľadu verifikátora neodlišuje od volania funkcií definovaných SystemVerilogu.

Hlavnou motiváciou pri zavádzaní tejto techniky bola vízia urýchlenia a postupnej automatizácie vytvárania verifikačného prostredia pre hardvérové jednotky generované systémom Codasip, ako aj verifikácie samotnej. Táto práca potvrdzuje, že je možné automaticky generovať časť implementácie *scoreboardu* spolu so zapojením externého výpočtu očakávaných výstupov v C++, čo predstavuje východiskový bod pre automatizáciu vytvárania ďalších častí verifikačného prostredia.

5.4 Funkčné pokrytie

Miera kompletnosti funkčnej verifikácie je odvodzovaná hlavne na základe funkčného pokrytia, preto je potrebné venovať problematike návrhu bodov pokrytia zvýšenú pozornosť.

Sledovanie všetkých možných hodnôt signálov s veľkou dátovou šírkou je značne neefektívne a často krát zbytočné. Vygenerovať všetky hodnoty 16 bitového signálu, aké má aj ALU, by trvalo generátoru pseudonáhodných čísel nezanedbateľné množstvo času. Navyše vo veľkom počte prípadov nemusí byť nutne preverená celá množina všetkých hodnôt signálu, ale stačí preveriť niekoľko zástupcov z množín podobných hodnôt. Preto je možné body pokrytia signálov väčšej dátovej šírky rozdeliť do tzv. *binov*. *Biny* predstavujú kontajnery hodnôt, ktoré sú označené za pokryté práve vtedy, ak bola pokrytá aspoň jedna z hodnôt nachádzajúcich sa v danom kontajneri. Ich použitím sa skvalitňuje sledovanie funkčného pokrytia zlúčením menej významných alebo naopak vyčlenením významnejších hodnôt do jednej množiny, javiacej sa pre mechanizmus zberu hodnôt sledovaného bodu pokrytia ako jedna hodnota.

Takto navrhnuté sledovanie funkčného pokrytia už síce dokáže sledovať, či boli otestované všetky význačné hodnoty, no vždy len izolovane od hodnôt ostatných signálov. Pre zistenie, či boli nastavené určité kombinácie hodnôt daných signálov naraz v jednom čase, alebo či sa vyskytla v priebehu testovania určitá sekvencia hodnôt pre daný signál, je nutné definovať zložitejšie body pokrytia.

Pre sledovanie hodnôt zasielaných na vstupný signál určujúci operáciu ALU je definovaný jednoduchý bod pokrytia, ktorý sleduje, či sa preskúšali všetky operácie. Ďalej sú definované rozsiahlejšie body pokrytia, ktoré v kombinácii s dobre navrhnutými obmedzeniami pre generátor vo veľkej

miere prispejú k zvýšeniu funkčného pokrytia verifikovanej jednotky. V nasledujúcich podkapitolách sú rozoberané navrhnuté body pokrytia a popísané obmedzenia definované pre generátor.

5.4.1 Vstupné operandy ALU

Body pokrytia pre operandy rozdeľujú množinu hodnôt každého operandu na 5 podmnožín. Dve z nich, implementované *binmi* `zeros` a `ones`, sú vyčlenené samostatne pre najmenšiu a najväčšiu možnú hodnotu v danom rozsahu, keďže práve tieto hodnoty môžu najčastejšie vyvolať nekorektné chovanie v chybnom hardvéri. Nemenej významné je sledovať výskyt krajných hodnôt ako vidno v definícii *binov* `small_values` a `big_values`. Ostatné hodnoty sú zoskupené v *bine* `others`. Obrázok 5.3 ukazuje deklaráciu bodu pokrytia pre sledovanie hodnôt zasielaných na vstup `ra` verifikovanej jednotky.

```
opA: coverpoint operandA {
      bins zeros = {0};
      bins ones = {16'hffff};
      bins small_values = {[16'h0001:16'h00ff]};
      bins big_values = {[16'hff00:16'hfffe]};
      bins others = {[16'h0100:16'hfeff]};
    }
```

Obrázok 5.3 Bod pokrytia pre hodnoty signálu `ra`.

5.4.2 Komplexné body pokrytia

Na obrázku 5.4 sú znázornené komplexné body pokrytia. Prvý z nich, `op_after_op`, je reprezentantom tzv. *transition coverage* a sleduje či nastali všetky možné sekvencie dvoch po sebe idúcich operačných kódov. Druhý, `cross_oper_opA_opB` je reprezentantom tzv. *cross coverage* a je vytvorený z jednoduchších, už skôr definovaných bodov. Sleduje, či boli nastavené všetky možné kombinácie vstupných hodnôt zasielaných na vstupné signály oboch operandov a všetkých operácií.

```
op_after_op: coverpoint operation {
      bins op_after_op [] = ([6'h00:6'h0e], [6'h10:6'h13],
                             [6'h17:6'h32], [6'h38:6'h39],
                              6'h3f
                              =>
                             [6'h00:6'h0e], [6'h10:6'h13],
                             [6'h17:6'h32], [6'h38:6'h39],
                              6'h3f); }
cross_oper_opA_opB: cross ex_op, opA, opB;
```

Obrázok 5.4 Zložitejšie body pokrytia.

5.4.3 Obmedzenia generátora

Pre dosiahnutie väčšej úrovne funkčného pokrytia je nutné definovať obmedzenia pre generátor náhodných hodnôt. Pre generovanie zmysluplných hodnôt pre vstupy verifikovanej jednotky sú definované nasledujúce obmedzenia. Ich zápis v jazyku SystemVerilog je na obrázku 5.5 a ich vplyv na veľkosť pokrytia je popísaná v podkapitole 6.3.2.

```
constraint values_range {
    operation inside {[6'h00:6'h0e], [6'h10:6'h13],
    [6'h17:6'h32], 6'h38, 6'h39, 6'h3f};
    operandA dist {16'h0000:/15, [16'h0001:16'h00ff]:/20,
    [16'h0100:16'hfeff]:/30, [16'hff00:16'hfffe]:/20,
    16'hffff:/15};
    operandB dist {16'h0000:/15, [16'h0001:16'h00ff]:/20,
    [16'h0100:16'hfeff]:/30, [16'hff00:16'hfffe]:/20,
    16'hffff:/15};
}
```

Obrázok 5.5 Obmedzenia pre generovanie hodnôt operandov a operácie.

Prvé obmedzenie pre členskú premennú vstupnej transakcie `operation`, obmedzuje generovanie hodnôt pre vstupný signál určujúci operáciu, ktorú má ALU vykonať, iba na špecifikáciou určené operačné kódy. Nasledujúce obmedzenia definované pre premenné `operandA` a `operandB` definujú obmedzenie pre hodnoty signálov vstupných operandov. Je pre ne definované váhové rozdelenie, ktoré rozdeľuje množinu ich hodnôt na podmnožiny. Každej podmnožine je priradená váha, ktorá určí početnosť výskytov hodnôt z danej podmnožiny pri generovaní hodnôt členských premenných transakcie. Zápis `[16'h0001:16'h00ff]:/20` znamená, že hodnoty z daného intervalu budú mať medzi ostatnými generovanými hodnotami 20% zastúpenie.

6 Priebeh testov a analýza výsledkov

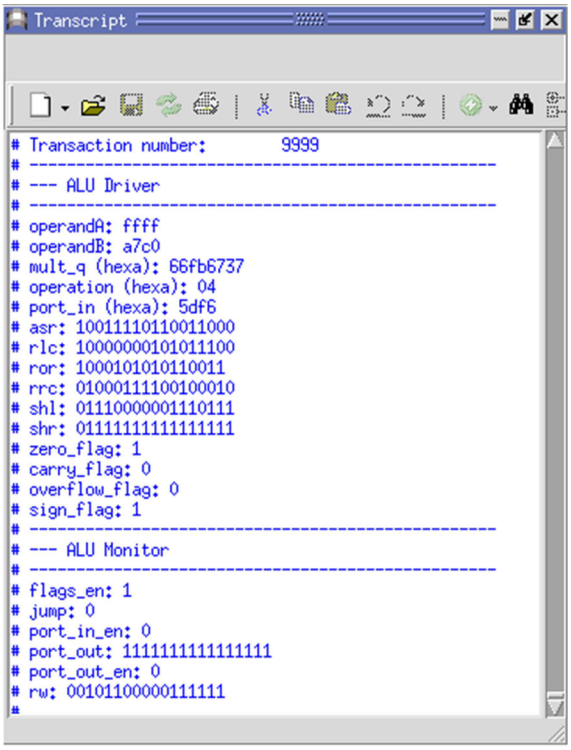
Kompilácia a simulácia verifikačného prostredia a verifikovanej jednotky je realizovaná pomocou nástroja ModelSim, v ktorom taktiež prebieha testovanie verifikovanej jednotky, zber štatistík o pokrytí a následná analýza výsledkov všetkých testov.

6.1 Spôsob testovania

Testovanie bolo založené na technike funkčnej verifikácie riadenej pokrytím (angl. *coverage driven verification*). Generátor vytváral jednotlivé stimuly pre verifikovanú jednotku podľa nastavených konštánt prostredia a definovaných obmedzení. Ak sa v priebehu testovania verifikácia zastavila v dôsledku chyby, bolo potrebné zanalyzovať doposiaľ získané výsledky a prípadne vytvoriť priame testy pre odhalenie možnej príčiny zastavenia verifikácie. Naopak po bezchybnom priebehu všetkých testov bola zahájená analýza štatistík funkčného pokrytia. Na základe jej výsledkov sa buďto upravili konštanty verifikačného prostredia, obmedzujúce podmienky pre generovanie stimulov, pridali ďalšie, podrobnejšie body pokrytia, alebo sa miera správnosti verifikovanej jednotky prehlásila za postačujúcu a verifikácia sa ukončila.

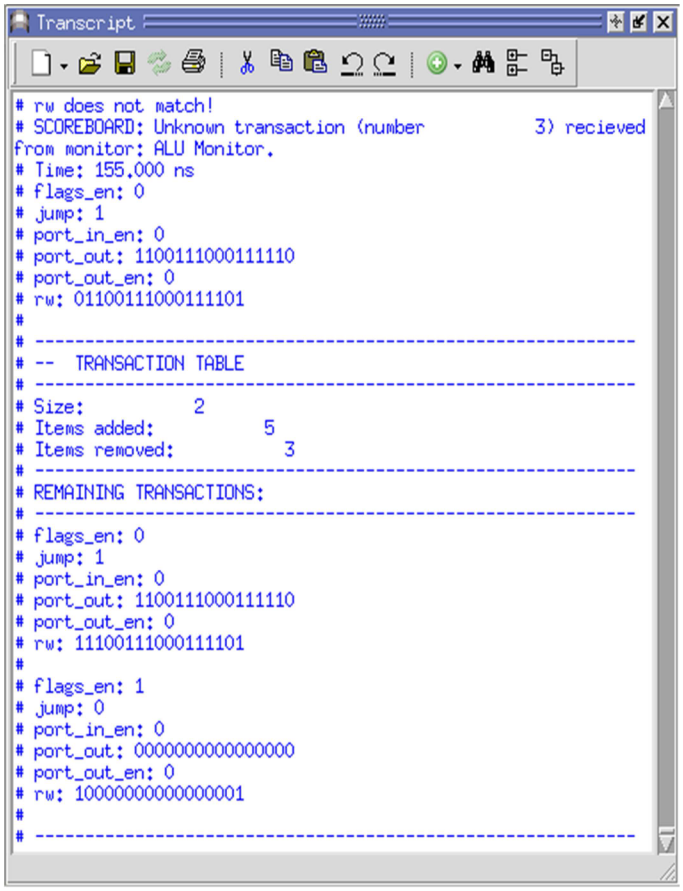
6.2 Interpretácia výstupov testov

V priebehu testovania sú generované výpisy o aktuálne zasielaných transakciách z *drivera* a prijímaných výstupných transakciách zachytených *monitorom*. Príklad časti výpisov je uvedený na obrázku 6.1.



Obrázok 6.1 Výpis posielaných transakcií.

Ak sa v priebehu testu vyskytne nezhoda pri porovnávaní transakcií v *scoreboarde*, priebeh testu sa zastaví a vypíše sa chybový výpis. Ten obsahuje transakciu prijatú z *monitora*, ktorá spôsobila nezhodu a zoznam jej dátových zložiek, ktorých hodnoty sú odlišné od hodnôt očakávaných. Takisto je zobrazený obsah transakčnej tabuľky, v ktorej sa nachádzajú prípadné ďalšie očakávané výstupné transakcie. Príklad chybového výpisu je uvedený na obrázku 6.2. Ak prebehnú všetky testy v poriadku, tzn. výstupy verifikovanej jednotky sú zhodné s predpovedanými výstupmi, tak je na konci testovacích výpisov uvedená informácia o miere funkčného pokrytia verifikovanej jednotky, určenej definovanými bodmi pokrytia. Príklad výpisu po úspešnom priebehu testov je na obrázku 6.3. Následná podrobná analýza štatistík funkčného pokrytia je uvedená v podkapitole 6.3.

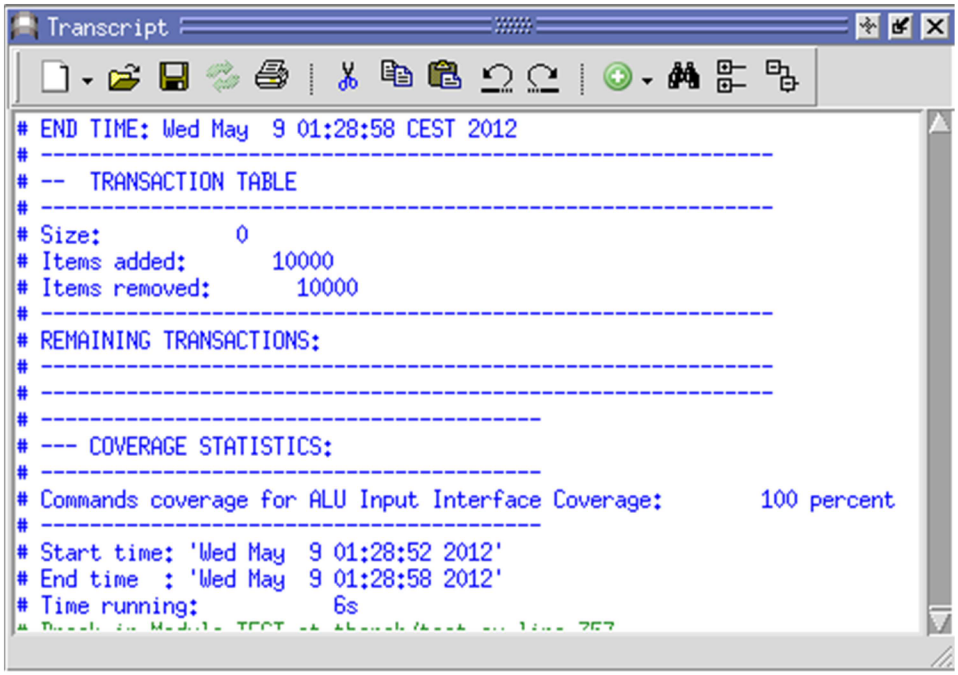


Obrázok 6.2 Výpis pri odhalení nezrovnalostí vo výstupných transakciách.

6.2.1 Odhalené chyby

V priebehu verifikácie bola odhalená chyba v návrhu prvej verzie verifikovanej jednotky, prejavujúca sa nesprávnym výpočtom *carry* pri operáciách, ktoré využívali aritmetické sčítanie. Táto chyba bola v novšom modeli už odstránená. Obe verzie ALU však neštandardne implementovali funkčnosť operácií aritmetických a logických posunov a rotácií, čo bolo spočiatku pokladané za chybu. Po konzultácii s návrhármi bolo zistené, že táto činnosť je vskutku správna a žiadaná. Verifikovaná ALU totiž tieto posuvy a rotácie nevykonáva, ale sú vykonávané inou funkčnou jednotkou v rámci procesora. Výsledky z nej sú potom privedené na dátové vstupy ALU, ktorá iba zabezpečí korektné nastavenie *carry* a presun danej predpočítanej hodnoty na výstup `rw`. Toto vysvetlenie však nič nemení na fakte, že pomocou verifikácie bola odhalená anomália v návrhu verifikovanej jednotky

spôsobená vyššie opísanou implementáciou. V prílohe 1 sú uvedené okomentované výpisy simulátora z priebehu testov, ktoré mali navodiť vyššie opísanú chybu a poukázať na zvláštne chovanie sa operácií posunov a rotácií.



Obrázok 6.3 Výpis po skončení simulácie, pri ktorej nebola odhalená chyba.

6.3 Analýza štatistík funkčného pokrytia

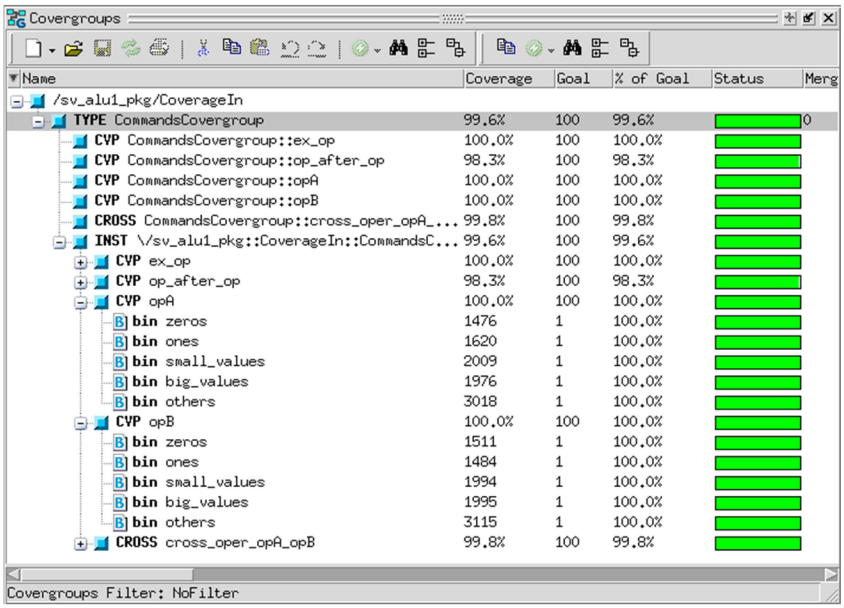
Okrem textových výpisov o stave funkčného pokrytia na konci simulačného behu, poskytuje ModelSim detailný grafický prehľad o naplnení jednotlivých definovaných bodov pokrytia, ako aj *binov*, z ktorých sú zložené. Obrázok 6.4 ukazuje okno ModelSimu pre detailný prehľad funkčného pokrytia.

Pre zvýšenie funkčného pokrytia a skvalitnenie jeho výpovednej hodnoty je ho možné vylepšiť viacerými spôsobmi. V nasledujúcich podkapitolách budú popísané možnosti, ako to dosiahnuť a ku každej z nich bude uvedené porovnanie výsledkov miery pokrytia pred a po aplikácii daného postupu v priebehu simulácie ALU procesora ADOP. Nastavenie verifikačného prostredia, ktoré je v nasledujúcich podkapitolách označené ako referenčné, zahŕňa zadefinovanie bodov pokrytia a obmedzujúcich podmienok pre generátor tak, ako sú popísané v podkapitole 5.4. Verifikácia s týmto nastavením ďalej označovaná ako referenčná. Všetky verifikačné behy, ktoré boli prevedené kvôli zberu hodnôt pre grafy v rámci tejto podkapitoly, boli prevedené na rovnakej sade testov s rovnako definovanými bodmi pokrytia ako v referenčnej verifikácii (*biny* sa ale môžu líšiť).

6.3.1 Zvýšenie počtu generovaných stimulov

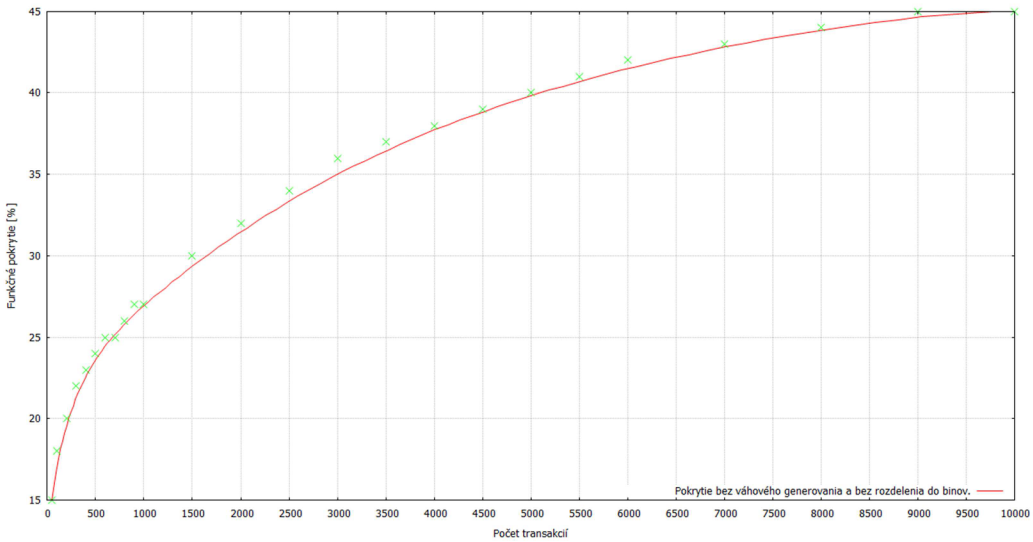
Najjednoduchší spôsob zvýšenia funkčného pokrytia je zvýšiť počet generovaných transakcií. Nasledujúci graf na obrázku 6.5 zobrazuje závislosť medzi počtom generovaných transakcií a hodnotou funkčného pokrytia pri verifikácii ALU ADOP. Zelené krížiky predstavujú namerané

hodnoty pokrytia pre daný počet transakcií, červená krivka potom ich aproximáciu Beziérovou krivkou. Toto značenie platí aj pre nasledujúce grafy. V priebehu merania bolo definované rozdelenie bodov pokrytia do explicitných *binov*, tzn. že sa nepoužil žiaden heuristický prístup pre tvorbu *binov* a každému z nich prislúchala práve jedna hodnota. Obmedzenia generátora pre váhové rozdelenie generovania vstupných hodnôt neboli definované.



Obrázok 6.4 Detailný prehľad funkčného pokrytia.

Z grafu je možné vidieť pomerne prudký nárast miery pokrytia pre nízke počty transakcií, ktorý sa ale zvyšovaním počtu generovaných stimulov znižuje. Toto znižovanie je možné vysvetliť opakovaním sa už raz vygenerovaných hodnôt pre dané body pokrytia, ktoré neprispievajú k ďalšiemu zvyšovaniu celkového pokrytia. Pokračovanie v zvyšovaní počtu transakcií nad určitú mieru je preto už značne nevýhodné napriek tomu, že vedie k želanému výsledku. Pre dosiahnutie aspoň 80% pokrytia je v tomto prípade nutné vygenerovať viac ako 200 000 transakcií, pričom pokrytie sa zvýši len o 35% na úkor viac než 20 krát dlhšieho času potrebného na jeho dosiahnutie.

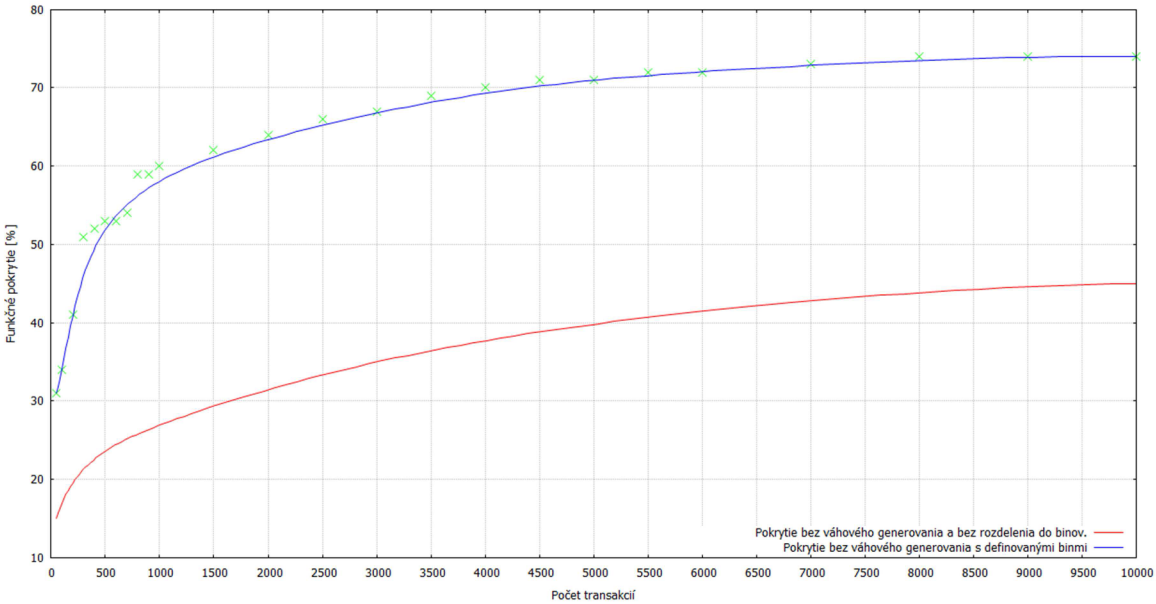


Obrázok 6.5 Závislosť funkčného pokrytia na počte generovaných transakcií.

6.3.2 Úprava bodov pokrytia a obmedzení generátora

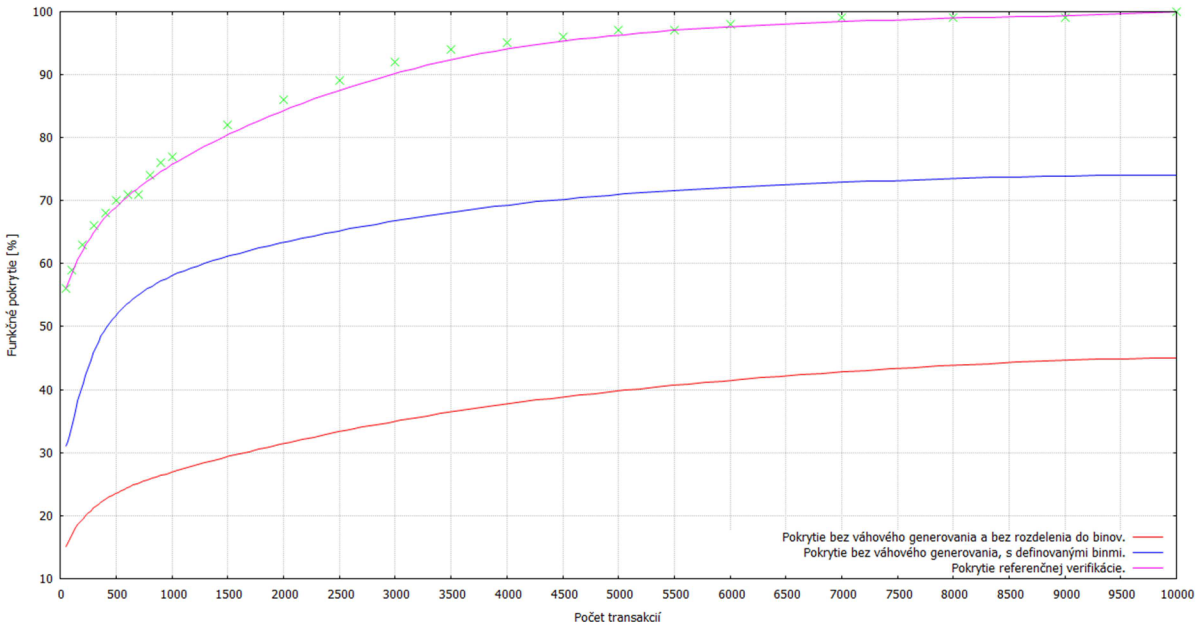
Pri verifikácii jednoduchých hardvérových jednotiek je už pri relatívne nízkom počte generovaných transakcií dosiahnuté maximálne pokrytie. V prípade zložitejších jednotiek, ku ktorým je vytvorená komplexná množina bodov pokrytia, nemusí zvyšovanie počtu generovaných stimulov stačiť pre dosiahnutie maximálneho pokrytia stavového priestoru danej jednotky. Navyše sa výrazne predlžuje doba verifikácie (viď predchádzajúca podkapitola), pri ktorej sa spotrebuje značné množstvo zdrojov. Preto je nutné aplikovať ďalšie postupy, pomocou ktorých je možné dosiahnuť maximálne pokrytie v čo najkratšom čase.

Na obrázku 6.6 sú zobrazené dva prípady závislosti funkčného pokrytia na počte generovaných transakcií. Závislosť zastúpená červenou krivkou je totožná so závislosťou z obrázku 6.5, tzn. že pri jej meraní boli sledované všetky možné hodnoty definovaných bodov pokrytia, a neboli definované obmedzenia generátora pre váhové rozdelenie generovania vstupných hodnôt. Modrá krivka znázorňuje vývoj funkčného pokrytia verifikácie, v ktorej bolo pre body pokrytia operandov dodefinované rozdelenie ich možných hodnôt do *binov* tak, ako je uvedené v referenčnej verifikácii. Takéto zoskupenie viacerých, pre verifikáciu podobných hodnôt prakticky spôsobí, že pri vygenerovaní jednej hodnoty z daného *binu* je do aktuálneho pokrytia pridaný stavový priestor jednotky, ktorý by bol inak pokrytý len v prípade, kedy by boli vygenerované všetky hodnoty z daného *binu*. Preto je možné v grafe už teraz vidieť takmer dvojnásobný nárast miery pokrytia tejto verifikácie oproti verifikácii, v ktorej boli sledované všetky možné hodnoty bodov pokrytia. Rozdelenie do *binov* síce nemusí preveriť všetky možné hodnoty daného bodu, ale význačné hodnoty zo všetkých *binov* sú otestované, čo je v prípade, že sú dobre navrhnuté, postačujúce.



Obrázok 6.6 Závislosť funkčného pokrytia na počte transakcií – rozdelenie bodov pokrytia do *binov*.

Na zvyšovaní miery funkčného pokrytia sa významne podieľajú aj obmedzenia pre generátor transakcií. Graf na obrázku 6.7 ukazuje závislosti z grafu na obrázku 6.6 (červená a modrá krivka). Fialová krivka zobrazuje priebeh funkčného pokrytia referenčnej verifikácie. Z grafu je možné vidieť, že dodefinovaním obmedzení pre generátor pre váhové rozdelenie generovania hodnôt vstupných signálov bolo dosiahnuté ďalšie zvýšenie miery funkčného pokrytia verifikácie, bez nutnosti zvýšiť počet generovaných transakcií, a tým pádom predĺžiť čas verifikácie.



Obrázok 6.7 Závislosť funkčného pokrytia na počte transakcií – porovnanie pokrytia pre jednotlivé prístupy

Ak by neboli použité uvedené prístupy na zlepšenie pokrytia a počet transakcií by bol stále zvyšovaný ako v prípade verifikácie v podkapitole 6.3.1, tak by čas potrebný pre dosiahnutie 100% pokrytia meraný v ráde minút výrazne prekračoval čas 6 sekúnd behu referenčnej verifikácie, kedy bolo dosiahnuté 100% pokrytie.

6.4 Porovnanie rýchlosti testov

V druhej, novšej verzii hardvérového modelu verifikovanej jednotky bol pre výpočet očakávaného výsledku v *scoreboarde* použitý externý výpočet cez DPI. Ako už bolo spomenuté v podkapitole 5.3, výpočet hodnôt výstupnej transakcie realizujú C++ funkcie automaticky generované z popisu obvodu v jazyku CodAL. V priebehu jednotlivých simulačných behov bol meraný čas potrebný pre vyhodnotenie testov pri použití výpočtu cez DPI rozhranie a takisto aj čas potrebný pre štandardné formovanie výslednej transakcie priamo v *scoreboarde* v jazyku SystemVerilog. Tabuľka 6.1 porovnáva dosiahnuté priemerné časy z piatich simulačných behov.

Počet transakcií	Výpočet cez DPI v C++	Výpočet bez DPI v SystemVerilogu
10 000	6s	6s
20 000	13s	12s
50 000	33s	29s

Tabuľka 6.1 Porovnanie trvania výpočtu.

Z tabuľky je vidieť, že výpočet cez DPI rozhranie je síce pomalší, no rozdiel nie je markantný, navyše už pri 10 000 transakciách bolo dosiahnuté funkčné pokrytie 100%. Spomalenie je pravdepodobne spôsobené réžiou volania externých funkcií výpočtu výstupov pomocou DPI rozhrania a nutnosťou namapovať dátové typy C++ na dátové typy SystemVerilogu. Ak by bol výpočet výsledkov zložitejší, napríklad u nejakej komplexnejšej jednotky, pravdepodobne by sa rozdiel časov

zmenšil a réžia volania výpočtu cez DPI by sa stala v takom prípade zanedbateľnou, prípadne by bol výpočet cez DPI dokonca rýchlejší.

Navyše nespornou výhodou zapojenia referenčných funkcií v C++ cez DPI rozhranie je fakt, že verifikátor nemusí odznova implementovať funkčnosť verifikovanej jednotky, aby mohol pristúpiť k verifikácii, čím sa urýchli tvorba verifikačného prostredia.

7 Záver

Cieľom práce bolo navrhnúť koncept verifikačného prostredia pre overenie správnosti generovaných funkčných jednotiek vo vývojovom prostredí Codasip a otestovať ho na niektorých exemplárnych jednotkách. Implementované verifikačné prostredie sa ukázalo byť vyhovujúce a schopné efektívne verifikovať generované jednotky. V priebehu verifikácie jednotky ALU procesora ADOP bola odhalená chyba v návrhu tejto jednotky, ktorá zasahovala viacero operácií podporovaných v ALU a navyše bolo upozornené na neštandardné chovanie sa niektorých operácií. Aj vďaka tomu sa podarilo poukázať na fakt, že začlenenie procesu funkčnej verifikácie do vývojového cyklu hardvéru, vyvíjaného v prostredí Codasip, je dobrý krok k jeho zefektívneniu.

Čo sa týka efektivity funkčnej verifikácie, tá samozrejme vzrastá, pokiaľ verifikátor podrobne zanalyzuje chovanie verifikovanej jednotky a na základe tejto analýzy vytvorí cielené body pokrytia a rozumné obmedzujúce podmienky pre generátor vstupných hodnôt. Po ukončení návrhu verifikačného prostredia pre ALU boli uskutočnené merania vplyvu jednotlivých bodov pokrytia a definovaných obmedzení pre generátor na veľkosť funkčného pokrytia. Vďaka tomu sa ich v rámci riešenia práce podarilo navrhnúť a upraviť tak, aby zabezpečovali rýchle a efektívne overenie funkčnosti verifikovanej jednotky na nízkom počte generovaných transakcií. Samozrejme, priestor na zlepšenie je tu stále a určite by sa bolo vhodné zamyslieť nad definovaním ešte komplexnejších bodov pokrytia pre obsiahnutie ešte väčšej časti stavového priestoru jednotky.

Zapojenie transformačných funkcií v C++ pre výpočet očakávaných výsledkov cez DPI rozhranie SystemVerilogu sa ukázalo byť výhodné z hľadiska urýchlenia verifikácie, ako aj urýchlenia samotnej tvorby verifikačného prostredia. Čas, ktorý bol predtým strávený implementáciou výpočtu výsledkov podľa špecifikácie, bolo možné venovať iným kľúčovým činnostiam v rámci verifikačného procesu, napríklad návrhu detailných bodov pokrytia. Ďalšou výhodou je, že takto realizovaný výpočet očakávaných výstupov vytvára priestor pre zautomatizovanie procesu tvorby *scoreboardu* verifikačného prostredia.

V rámci budúceho vývoja práce by sa mohlo pristúpiť k čiastočnému zautomatizovaniu vytvárania i ďalších častí verifikačného prostredia a napevno tým začleniť verifikáciu do vývojového cyklu návrhu hardvéru prostredníctvom systému Codasip. Určite bude možné úplne zautomatizovať generovanie bázových tried prostredia, tried pre vstupné a výstupné transakcie, SystemVerilogového rozhrania pre DUT a modulu predstavujúceho DUT. Pokiaľ bude vždy k dispozícii aj referenčný výpočet alebo referenčný model verifikovanej jednotky, ktorý bude možné zapojiť do verifikačného prostredia napríklad cez DPI rozhranie, bude možné zautomatizovať aj tvorbu celej štruktúry *scoreboardu*. Za zváženie stojí možnosť generovať *driver* a *monitor*, pre ktoré by sa musel dodefinovať spôsob, ako im predať informáciu o čase, v ktorom majú zasielať/prijímať transakcie do/z DUT. V prípade automatického generovania triedy pre sledovanie funkčného pokrytia by už pravdepodobne nastal problém, pretože pre vytváranie zložitejších bodov pokrytia je potrebné sa zamyslieť nad zmyslom sledovaných signálov a ich závislosťou na nastavení iných signálov.

Ďalším vhodným zlepšením by mohlo byť vystavanie verifikačného prostredia nad niektorou z novších metodík (napr. UVM), ktoré poskytujú vylepšenú funkčnosť a väčšiu flexibilitu.

Literatúra

[1] BERGERON, Janick. *Writing testbenches using System Verilog*. New York: Springer, c2006, 412 s. ISBN 03-872-9221-7.

[2] MEYER, Andreas. *Principles of functional verification*. Amsterdam: Elsevier, c2003, 206 s. ISBN 07-506-7617-5.

[3] SPEAR, Chris. *SystemVerilog for verification*: *a guide to learning the testbench language features*. New York, NY: Springer, c2006, 301 s. ISBN 03-872-7038-8.

[4] BERGERON, Janick. CERNY, Eduard. HUNTER, Alan. NIGHTINGALE, Andrew: *Verification methodology manual for SystemVerilog*. Springer, USA, 2005. ISBN 0387-25556-7

[5] MOLINA, A., CADENAS, O.: *Functional verification: Approaches and challenges*. Latin America Applied Research, 2007.
Dostupné na URL: http://www.scielo.org.ar/pdf/laar/v37n1/v37n1a13.pdf

[6] Kolektív autorov: *Codasip manual*. Codasip

[7] Kolektív autorov: *CodAL manual.* Codasip

[8] ACCELLERA. *SystemVerilog* [online]. [cit. 2012-05-13]. Dostupné z: www.systemverilog.org

[9] DOULOS. *The Guide to SystemVerilog* [online]. [cit. 2012-05-13]. Dostupné z: http://www.doulos.com/knowhow/sysverilog/

[10] Marcela Šimková: Hardware Accelerated Functional Verification, diplomová práce, Brno, FIT VUT v Brně, 2011

[11] Marcela Šimková: Prostředí pro verifikaci DMA řadičů v jazyku SystemVerilog, bakalářská práce, Brno, FIT VUT, v Brně, 2009

[12] IEEE STD 1800-2009. *IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language*. 2009. Dostupné z: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5354133

[13] MENTOR GRAPHICS. *Verification Academy* [online]. [cit. 2012-05-13]. Dostupné z: www.verificationacademy.com

Príloha 1

Súhrn odhalených chýb a podozrivých operácií v ALU ADOP s niekoľkými príkladmi ich výskytu.

```
###########
!PLATÍ IBA PRE STARŠÍ MODEL!
Operácie, ktoré obsahujú aritmetické sčítanie nesprávne nastavujú
carry - vždy na nulu. Sú to:
ADD, ADC, INC, CALL, JZ, JNZ, JC, JNC, JS, JNS, JO, JNO, JGE,
JLT, JGT, JLE, JAT, JBE, JMP

# ************************************************************
# ADD operácia
# ************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
# operandA: 0100001010111100  <----
# operandB: 1111111111111111  <----
  mult_q: 2667408295
# operation (hexa): 01  <----
  port_in: 24588
  asr: 10010011100001011
  rlc: 10000101010010001
  ror: 0000000100001011
  rrc: 10000001011101101
  shl: 01110100001110110
  shr: 01111011111110110
  zero_flag: 0
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
```

```
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 0100001010111100
  port_out_en: 0
# rw: 00100001010111011  <----
# ************************************************************
# CALL operácia
# ************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
# operandA: 1100111000111110  <----
# operandB: 1111111111111111  <----
  mult_q: 3625207061
# operation (hexa): 1d  <----
  port_in: 37562
  asr: 00000110001010101
  rlc: 11111000000001110
  ror: 0010110000001011
  rrc: 00010100100101110
  shl: 00100001111110001
  shr: 00101110101010011
  zero_flag: 1
  carry_flag: 0
  overflow_flag: 1
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 0
  jump: 1
  port_in_en: 0
  port_out: 1100111000111110
  port_out_en: 0
# rw: 01100111000111101  <----
```

```
# ****************************************************************
# ADC operácia
# ****************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
# operandA: 1111111111111111  <----
# operandB: 1111111111111111  <----
  mult_q: 2677909095
# operation (hexa): 02  <----
  port_in: 16338
  asr: 10110101100100110
  rlc: 10100011111011011
  ror: 1001100101101100
  rrc: 01001100011100011
  shl: 01010110111000100
  shr: 01011001101101001
  zero_flag: 0
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 1111111111111111
  port_out_en: 0
# rw: 0111111111111110  <----
      ^
# ****************************************************************
# INC operácia
# ****************************************************************
  ------------------------------------------------
  --- ALU Driver
```

```
  ------------------------------------------------
# operandA: 1111111111111111  <----
# operandB: 0000000000000000  <----
  mult_q: 1130308030
# operation (hexa): 12  <----
  port_in: 49562
  asr: 10110011111010000
  rlc: 10100000000111100
  ror: 1100010101000011
  rrc: 01010111100011010
  shl: 01010100110111111
  shr: 01011010110110111
  zero_flag: 0
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 1111111111111111
  port_out_en: 0
# rw: 00000000000000000  <----

###########
!PLATÍ PRE OBA MODELY!
Operácie posunov a rotácií sú implementované veľmi neštandardne
a podozrivo.
Niektoré rotácie a posuny, konkrétne SHL, RRC a RCL, iba
prepisujú príslušný vstup na výstup, pričom nevykonajú žiadnu
zmenu dát.
SHR a ASR vstup zrotujú o jeden bit doprava a pošlú na výstup.
Operácia ROR má ako jediná 16 bitový vstup
(ostatné majú 17 bitový, a takisto výstup ma 17 bitov) a prevedie
```

```
iba znamienkové rozšírenie vstupu na výstup.

###################### Podozrivé operácie ######################

# ************************************************************
# SHL operácia - vstup na t_shl sa neposunie, iba sa prepíše na
# výstup rw.
# ************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 1111111111111111
  operandB: 0000000000000000
  mult_q: 1885334364
  operation (hexa): 09
  port_in: 62332
  asr: 01100010001010100
  rlc: 00000011000101111
  ror: 1101001000110101
  rrc: 11100110101000111
# shl: 11011101111110110  <----
  shr: 11010000100011101
  zero_flag: 1
  carry_flag: 0
  overflow_flag: 1
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 1111111111111111
  port_out_en: 0
# rw: 11011101111110110 <----
```

```
# ************************************************************
# SHR operácia - vstup na t_shr sa neposunie, ale prevedie sa
# rotácia o jedno doprava.
# ************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 0100001010111100
  operandB: 1111111111111111
  mult_q: 2667408295
  operation (hexa): 0a
  port_in: 24588
  asr: 10010011100001011
  rlc: 10000101010010001
  ror: 0000000100001011
  rrc: 10000001011101101
  shl: 01110100001110110
# shr: 01001101011110111  <----
  zero_flag: 0
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 0100001010111100
  port_out_en: 0
# rw: 10100110101111011  <----

# ************************************************************
# ASR operácia - vstup na t_asr sa aritmeticky neposunie, ale
# prevedie sa rotácia o jedno doprava.
# ************************************************************
```

```
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 0000000000000000
  operandB: 0000000000000000
  mult_q: 3312128972
  operation (hexa): 0b
  port_in:  6893
# asr: 11111101111111000  <----
  rlc: 11001001100010010
  ror: 0111011100011110
  rrc: 00111101000101100
  shl: 00111101101000000
  shr: 00110001101100010
  zero_flag: 0
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 0
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 0000000000000000
  port_out_en: 0
# rw: 01111110111111100  <----

# ************************************************************
# ROR operácia - vstup na t_ror sa nerotuje, prevedie sa jeho
# znamienkové rozšírenie o jeden bit. Totiž vstup t_ror je 16
# bitový a výstup rw je 17 bitový. Operácia iba skopíruje vstup
# na výstup a MSB výstupu (rw[16]) nastaví podľa MSB vstupu
# t_ror.
# ************************************************************
  ------------------------------------------------
```

```
  --- ALU Driver
  ------------------------------------------------
  operandA: 0000000000000000
  operandB: 1111111111111111
  mult_q:  393171169
  operation (hexa): 0c
  port_in: 21191
  asr: 10000110010100101
  rlc: 10001001010100111
# ror: 1010101111111000  <----
  rrc: 01110000001111000
  shl: 01100101111100001
  shr: 01101000100100011
  zero_flag: 1
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 0000000000000000
  port_out_en: 0
# rw: 11010101111111000  <----

  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 1111111111111111
  operandB: 1111111111111111
  mult_q:  730269621
  operation (hexa): 0c
  port_in: 17139
  asr: 11100001101011111
```

```
  rlc: 00110111110100110
# ror: 0101101111111000  <----
  rrc: 11011101011110101
  shl: 11100010000011000
  shr: 11101101011101100
  zero_flag: 0
  carry_flag: 1
  overflow_flag: 1
  sign_flag: 0
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 1111111111111111
  port_out_en: 0
# rw: 00101101111111000  <----


# ************************************************************
# RLC a RRC operácia - iba prepíše vstup t_rlc/t_rrc na
# výstup.
# ************************************************************
  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 1111111111111111
  operandB: 0111110111000010
  mult_q: 3086902761
# operation (hexa): 0d    <-- RLC
  port_in: 19174
  asr: 00010101000110000
# rlc: 01111101011011000  <----
  ror: 0000101010010010
  rrc: 10001001111110101
```

```
  shl: 10011101011110000
  shr: 10010011101111100
  zero_flag: 1
  carry_flag: 0
  overflow_flag: 0
  sign_flag: 1
  ------------------------------------------------
  --- ALU Monitor
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 1111111111111111
  port_out_en: 0
# rw: 01111101011011000  <----

  ------------------------------------------------
  --- ALU Driver
  ------------------------------------------------
  operandA: 0000000000000000
  operandB: 0000000000000000
  mult_q: 3772897807
# operation (hexa): 0e    <-- RRC
  port_in: 58076
  asr: 11001101001010101
  rlc: 00111101011011011
  ror: 0001100000101011
# rrc: 00110111000001101  <----
  shl: 00000110010010110
  shr: 00000111000011011
  zero_flag: 1
  carry_flag: 1
  overflow_flag: 1
  sign_flag: 0
  ------------------------------------------------
  --- ALU Monitor
```

```
  ------------------------------------------------
  flags_en: 1
  jump: 0
  port_in_en: 0
  port_out: 0000000000000000
  port_out_en: 0
# rw: 00110111000001102  <----
```

Príloha 2

Obsah priloženého CD

- Text práce vo formáte PDF.
- Návod pre kompiláciu a spustenie verifikácie.
- Verifikačné prostredia pre verifikáciu oboch modelov ALU.

Návod je uložený v textovom súbore `readme.txt`. Jednotlivé verifikácie sú uložené v zložkách `old` (starší model) a `new` (nový model). Ich vnútorná adresárová štruktúra je s malými odchýlkami zhodná. Obe obsahujú podzložky `src` a `ver`. V `src` je uložený VHDL model verifikovanej jednotky. Zložka `ver` obsahuje skript pre preklad verifikačného prostredia a spustenie verifikácie `alu.fdo` a súbor pre zobrazenie signálov v ModelSim-e `signals.fdo`. Ďalej je tu zložka `tbench` so súbormi tried komponentov verifikačného prostredia. Novší model v nej má navyše implementáciu (súbor `main_alu_alu_1_behavior.cpp`) a knižnicu s importovanými DPI funkciami `main_alu_alu_1_behavior.so` a ich SystemVerilogové rozhranie v súbore `main_alu_alu_1_behavior.sv`. Okrem toho je v koreňovom adresári CD zložka s bázovými triedami verifikačného prostredia `ver_base`.

```
/Funkčná verifikácia výpočtových jednotiek procesoru.pdf
/readme.txt
/old
  /src
    /main_alu_alu_1_behavior.vhd
  /ver
    /alu.fdo
    /signals.fdo
  /tbench
    /alu1_coverage.sv
    /alu1_driver.sv
    /alu1_ifc.sv
    /alu1_input_transaction.sv
    /alu1_monitor.sv
    /alu1_output_transaction.sv
    /dut.sv
    /scoreboard.sv
    /sv_alu1_pkg.sv
    /test.sv
    /test_pkg.sv
    /testbench.sv
/new
  /src
    /main_alu_alu_1_behavior_t.opt.vhd
  /ver
    /alu.fdo
    /signals.fdo
```

```
  /tbench
    /alu1_coverage.sv
    /alu1_driver.sv
    /alu1_ifc.sv
    /alu1_input_transaction.sv
    /alu1_monitor.sv
    /alu1_output_transaction.sv
    /dut.sv
    /main_alu_alu_1_behavior.cpp
    /main_alu_alu_1_behavior.so
    /main_alu_alu_1_behavior.sv
    /scoreboard.sv
    /sv_alu1_pkg.sv
    /test.sv
    /test_pkg.sv
    /testbench.sv
/ver_base
  /base.fdo
  /driver.sv
  /generator.sv
  /input_cbs.sv
  /math_pkg.sv
  /monitor.sv
  /output_cbs.sv
  /sv_common_pkg.sv
  /transaction.sv
  /transaction_table.sv
```

Príloha 3

Zoznam operácií ALU ADOP

Ak nie je určené v popise operácie inak, všetky operácie prepisujú vstup z `ra` na výstup `port_out` a ostatné výstupné signály sú nastavené do 0. Pri operáciách ALU, ktoré používajú aritmetické operácie, je do MSB `rw` uložený prípadný *carry/borrow*, inak je nastavený na 0. Niektoré operácie predstavujú iba časť skutočnej operácie, pretože pri ich činnosti je potrebné si uložiť stav priebehu operácie, ale samotná ALU vyňatá z procesora je len kombinačný obvod.

Operačný kód	Operácia	Popis
0x00	MOV	Presunie druhý operand na výstup. `rw = rb`

0x01	ADD	Sčíta operandy a nastaví výstup indikujúci zmenu príznakov.
		`rw = ra + rb`
		`flags_en = 1`

0x02	ADC	Sčíta operandy s *carry*, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra + rb + cf`
		`flags_en = 1`

0x03	SUB	Odčíta operandy, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra - rb`
		`flags_en = 1`

0x04	SBB	Odčíta operandy s *borrow*, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra - (rb + cf)`
		`flags_en = 1`

0x05	AND	Logický súčin operandov, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra & rb`
		`flags_en = 1`

0x06	OR	Logický súčet operandov, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra | rb`
		`flags_en = 1`

0x07	XOR	Exkluzívny súčet operandov, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra ^ rb`
		`flags_en = 1`

0x08	CMP	Porovná operandy, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra - rb`
		`flags_en = 1`

0x09	SHL	Všetky operácie posuvov a rotácií sú vykonané inou funkčnou jednotkou v rámci procesora, ALU by mala iba previesť patričné vstupy na výstup `rw`. Logický posun doľava: `rw = t_shl`

0x0A	SHR	Popis viď SHL. Logický posun doprava.
		`rw = t_shr >> 1`
		`rw[16] = t_shr[0]`

0x0B	ASR	Popis viď SHL. Aritmetický posun doprava.
		`rw = t_asr >> 1`
		`rw[16] = t_asr[0]`

0x0C	ROR	Popis viď SHL. Rotácia doprava.
		`rw = t_ror`
		`rw[16] = t_ror[15]`

0x0D	RLC	Popis viď SHL. Rotácia doľava cez *carry*. `rw = t_rlc`
0x0E	RRC	Popis viď SHL. Rotácia doprava cez *carry*. `rw = t_rrc`
0x10	NOT	Prevedie jednotkový doplnok prvého operandu. `rw = ~ra`

0x11	NEG	Prevedie dvojkový doplnok prvého operandu, nastaví výstup indikujúci zmenu príznakov.
		`rw = 0 - ra`
		`flags_en = 1`

0x12	INC	Inkrementuje prvý operand, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra + 1`
		`flags_en = 1`

0x13	DEC	Dekrementuje prvý operand, nastaví výstup indikujúci zmenu príznakov.
		`rw = ra - 1`
		`flags_en = 1`

0x17	POPF	Nastavenie príznakov uložených v zásobníku. `flags_en = 1`
0x18	CLC	Vymazanie príznaku *carry*. `flags_en = 1`
0x19	STC	Nastavenie príznaku *carry*. `flags_en = 1`
0x1A	CLI	Vymazanie príznaku *interrupt*. `flags_en = 1`
0x1B	STI	Nastavenie príznaku *interrupt*. `flags_en = 1`

0x1C	IRC	Volanie obsluhy prerušenia.
		`rw = rb`
		`flags_en = 1`
		`jump = 1`

0x1D	CALL	Volanie procedúry.
		`rw = ra + rb`
		`jump = 1`

0x1E	RET	Návrat z procedúry.
		`rw = rb`
		`jump = 1`

0x1F	RETI	Návrat z obsluhy prerušenia.
		`rw = rb`
		`flags_en = 1`

		`jump = 1`

0x20	JZ	Skoč, ak je nastavený `zf`.
		`rw = ra + rb`
		`jump = zf`

0x21	JNZ	Skoč, ak nie je nastavený `zf`.
		`rw = ra + rb`
		`jump = ~zf`

0x22	JC	Skoč, ak je nastavený `cf`.
		`rw = ra + rb`
		`jump = cf`

0x23	JNC	Skoč, ak nie je nastavený `cf`.
		`rw = ra + rb`
		`jump = ~cf`

0x24	JS	Skoč, ak je nastavený `sf`.
		`rw = ra + rb`
		`jump = sf`

0x25	JNS	Skoč, ak nie je nastavený `sf`.
		`rw = ra + rb`
		`jump = ~sf`

0x26	JO	Skoč, ak nie je nastavený `ovf`.
		`rw = ra + rb`
		`jump = ovf`

0x27	JNO	Skoč, ak nie je nastavený `ovf`.
		`rw = ra + rb`
		`jump = ~ovf`

0x28	JGE	Skoč, ak bolo väčšie alebo rovné.
		`rw = ra + rb`
		`jump = (sf == ovf )`

0x29	JLT	Skoč, ak bolo menšie.
		`rw = ra + rb`
		`jump = (sf != ovf)`

0x2A	JGT	Skoč, ak bolo väčšie.
		`rw = ra + rb`
		`jump = (sf == of && !zf)`

0x2B	JLE	Skoč, ak bolo menšie alebo rovné.
		`rw = ra + rb`
		`jump = (sf != ovf || zf)`

0x2C	JAT	Skoč, ak bolo nad. (unsigned)
		`rw = ra + rb`
		`jump = (!zf && !cf)`

0x2D	JBE	Skoč, ak bolo nad alebo rovno. (unsigned)
		`rw = ra + rb`
		`jump = (zf || cf)`

0x2E	JMP	Nepodmienený relatívny skok.
		`rw = ra + rb`
		`jump = 1`

0x2F	JMPA	Nepodmienený absolútny skok.

		`rw = rb` `jump = 1`

0x30	MULU	Pomocná operácia pri násobení.
		`rw = mult_q`
		`flags_en = 1`

0x31	MULS	Pomocná operácia pri násobení.
		`rw = mult_q`
		`flags_en = 1`

0x32	MHR	Pomocná operácia pri násobení. `rw = (mult_q >> 16)`

0x38	IN	Presun dát z `port_in` na výstup.
		`rw = port_in`
		`port_in_en = 1`

0x39	OUT	Dáta pripravené na `port_out`. `port_out_en = 1`
0x3F	HALT	