VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

LADĚNÍ SOFTWARE V CODASIP STUDIU POMOCÍ JTAG ROZHRANÍ SIMULOVANÉM V RTL SIMULÁTORU

SOFTWARE DEBUGGING IN CODASIP STUDIO USING JTAG INTERFACE SIMULATED IN RTL SIMULATOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **KAMIL MICHL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19725/2016/xmichl02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Michl Kamil**
Obor: Informační technologie
Téma: **Ladění software v Codasip Studiu pomocí JTAG rozhraní simulovaném v RTL simulátoru**
Software Debugging in Codasip Studio Using JTAG Interface Simulated in RTL Simulator
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s RTL simulátory a s API DPI, VPI a VHPI.
2. Seznamte se se standardem JTAG.
3. Seznamte se se způsobem ladění software pomocí Codasip Studia a s API, které Codasip Studio nabízí.
4. Navrhněte způsob propojení API Codasip Studia s vybraným API RTL simulátoru tak, aby šlo ladit software na procesoru, který je simulován v RTL simulátoru.
5. Návrh implementujte a otestujte na několika procesorech dle konzultace s vedoucím.
6. Zhodnoťte efektivitu implementace a možná rozšíření.

Literatura:

- Codasip Studio User Guide.
- Questa User Guide.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 4 zadání.

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

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

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

Vedoucí: **Zachariášová Marcela, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato práce se zabývá možností propojení RTL simulace procesoru se softwarovým debuggerem. Podle mého návrhu probíhá komunikace mezi těmito komponentami přes JTAG a Nexus rozhraní. Simulace je ovládána pomocí vybraného rozhraní mezi jazyky pro popis hardwaru a softwaru. Pro implementaci je použit JTAG adaptér od společnosti Codasip, RTL simulátor Questa Advanced Simulator od společnost Mentor, a Siemens Business, a rozhraní VPI pro komunikaci mezi jazyky Verilog a C++. Teoretická a částečně i praktická část této práce je použitelná pro více možných implementací zavislých na rozdílných programech a rozhraních. Konkrétní implementace uvedená v této práci je otestována a je funkční. V současnosti je používána společností Codasip a bude se pravděpodobně v budoucnu rozvíjet a vylepšovat.

Abstract

This thesis is dealing with an option to connect the RTL simulation of a processor with a software debugger. Acording to my design, the communication between these components is handled using the JTAG and the Nexus interface. The simulation is controlled by a selected interface between hardware and software description languages. For the implementation, following components are used: JTAG adapter created by Codasip, RTL simulator Questa Advanced Simulator created by Mentor, a Siemens Business, and VPI interface for communication between Verilog and C++ languages. Concept presented in this thesis can be used on other implementations that depend on different programs and interfaces. The implementation contained in this thesis was tested and is fully functional. Nowadays, it is used by Codasip company and it will probably be updated and enhanced in the future.

Klíčová slova

JTAG rozhraní, RTL simulace, VPI rozhraní, Nexus rozhraní, Codasip, Questa Advanced Simulator

Keywords

JTAG interface, RTL simulation, VPI interface, Nexus interface, Codasip, Questa Advanced Simulator

Citace

MICHL, Kamil. *Ladění software v Codasip Studiu pomocí JTAG rozhraní simulovaném v RTL simulátoru*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zachariášová Marcela.

Ladění software v Codasip Studiu pomocí JTAG rozhraní simulovaném v RTL simulátoru

Prohlášení

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

. . . . . .
Kamil Michl
15. května 2017

Poděkování

Děkuji za výbornou spolupráci všem zaměstnancům společnosti Codasip. Zvláště pak Ing. Marcele Zachariášové Ph.D. a Ing. Zdeňku Přikrylovi Ph.D.

Obsah

Kapitola 1

Úvod

Cílem této práce je navrhnout a implementovat propojení JTAG (angl. zk. *Joint Test Action Group*) adaptéru, Codasip Studia a vybraného RTL (angl. zk. *Register Transfer Level*) simulátoru. Toto propojení by mělo umožnit současné ladění softwaru a hardwarového modelu, na kterém software běží. Výhodou tohoto řešení je, že daný hardware nemusí být vyroben, ale dá se použít pouze jeho model a jeho simulace pro ladící účely. Toho lze dosáhnout pomocí hardwarových simulátorů a jejich rozhraní, které umožňují za běhu kontrolovat a sledovat simulaci. Knihovna JTAG adaptér od společnosti Codasip umožňuje přeposílání dat z ladícího programu (často Codasip Studio) do připojeného hardwaru a zpět pomocí různých komunikačních metod, které jsou v JTAG adaptéru implementovány jako samostatné pluginy.

Výsledkem této práce by měl být návrh aplikace, která je schopná napojit se na konkrétní RTL simulátor a na program typu debugger, který bude vhodným způsobem posílat data. Dále by měla být výsledkem konkrétní implementace tohoto návrhu, která bude používat JTAG adaptér a Codasip Studio jako debugger a bude připojena k simulátoru Questa Advanced Simulator od společnosti Mentor, a Siemens Business. To umožní uživateli, aby ladil program v jazyce C nebo assembler přeložený pro konkrétní hardware, a zároveň aby mohl kdykoliv zjistit nebo změnit stav daného hardwaru uvnitř simulace přímo při vykonávání programu.

1.1 Obsah kapitol

Ve druhé kapitole této práce jsou popsány vybrané používané RTL simulátory a komunikační rozhraní mezi těmito simulátory a vybranými jazyky pro popis hardwaru. Některé z těchto simulátorů a rozhraní budou v této práci využity pro vytvoření hardwarové simulace a napojení na ni.

Ve třetí kapitole je popsáno JTAG rozhraní, které bude ve výsledném programu sloužit k přenášení jednotlivých bitů dat mezi JTAG adaptérem a simulací. Také je zde okrajově zmíněno Nexus rozhraní, které zajišťuje vykonávání příkazů poslaných přes JTAG rozhraní. V poslední části kapitoly je popsán JTAG adaptér a rozhraní, které adaptér používá na připojení svých pluginů. Toto rozhraní bude muset výsledný program implementovat.

Ve čtvrté kapitole je podrobně popsán návrh a implementace mého řešení. V implementaci se využívá rozhraní VPI (angl. zk. *Verilog Procedural Interface*) pro Questa Advanced Simulator a také rozhraní JTAG adaptéru.

V páté kapitole se zabývám především prerekvizitami pro správnou funkčnost programu a také jeho testováním. Popisuji zde i příklad debuggeru dostupného v produktu Codasip Studio od společnosti Codasip, který lze použit pro ovládání simulace uživatelem.

V poslední, šesté, kapitole shrnuji své poznatky. Poukazuji na výhody i nedostatky mé implementace a uvádím její praktické využití.

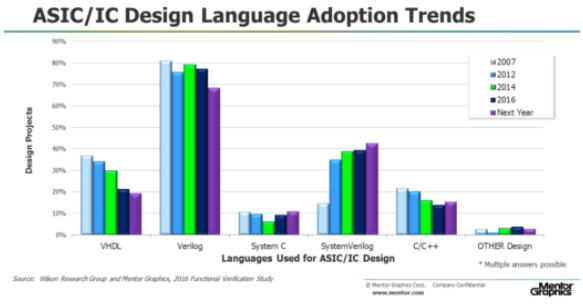
Kapitola 2

Simulátory a komunikační rozhraní

V této kapitole jsou popsány dostupné RTL simulátory a jejich komunikační rozhraní.

2.1 RTL simulátory

RTL simulátory jsou programy, které dokáží simulovat chování hardwaru popsaného pomocí RTL úrovně popisu. Pro takovýto popis se používají především jazyky Verilog a VHDL, ale můžeme se setkat i s jinými jazyky jako například SystemC nebo SystemVerilog. Dle studie provedené v roce 2016 je nejpoužívanějším jazykem Verilog a mezi předně používané jazyky se dostává SystemVerilog na úkor VHDL (viz obrázek 2.1). RTL úroveň abstrakce je určena primárně pro popis chování jednotlivých prvků. Neobsahuje informace o tom, jaký typ hardwaru nebo logického obvodu je použit pro realizaci designu. To umožňuje jednodušší návrh hardwaru především pro uživatele méně znalé v oblasti logických obvodů. Základním principem RTL popisu je deklarace použitých registrů a jiných komponent zajišťujících především aritmetické a logické funkce a implementace jejich chování.



Obrázek 2.1: Využití jazyků pro popis hardwaru
[15]

RTL simulátory dokáží načíst a zpracovat validní RTL popis obvodu společně s připraveným testovacím prostředím (někdy označováno jako testbench) a takovouto simulaci sledovat. Dále umožňují ovládat jednotlivé signály uvnitř RTL modelu a zároveň číst jejich hodnoty. Mezi základní funkce RTL simulátorů patří nahrání a příprava RTL úrovně popisu, spuštění, pozastavení a ukončení simulace, zjištění hodnoty libovolného signálu v jakémkoliv simulačním čase a změna hodnoty libovolného signálu v aktuálním čase. RTL simulátory implementují více funkcionalit, které usnadňují práci a ladění, např. optimalizace RTL

popisu před spuštěním simulace, plánování změn signálu v předstihu a podobně. Tyto funkce však nejsou z pohledu této práce důležité, a proto nebudou dále rozvedeny a popsány.

Podrobněji zde boudou zmíněny pouze dva RTL simulátory, které mám k dispozici. Těmito simulátory jsou Questa Advanced Simulator od společnosti Mentor, a Siemens Business, a Riviera-PRO od společnosti Aldec. Oba tyto simulátory implementují základní funkcionality zmíněné dříve a jsou v současnosti používány v industriální sféře.

2.1.1 Questa Advanced Simulator

Questa Advanced Simulator je RTL simulátor od společnosti Mentor. Podporuje jazyky Verilog, SystemVerilog, VHDL i SystemC pro popis hardwaru. Primárně je tento simulátor součástí produktu Mentor Enterprise Verification Platform, který je používán pro verifikaci hardwarových modelů. Lze jej však použít i samostatně pro simulaci a ladění uživatelem vytvořeného hardwarového modelu. Od společnosti Mentor je znám rovněž simulátor ModelSim, který je simulátoru Questa Advanced Simulator velice podobný, neobsahuje ovšem všechny jeho funkcionality. Pro přístup k simulaci z jiných programovacích jazyků poskytuje Questa Advanced Simulator komunikační rozhraní VPI pro jazyk Verilog, FLI (angl. zk. *Foreign Language Interface*) pro jazyk VHDL a SystemVerilog DPI (angl. zk. *Direct Programming Interface*) pro jazyk SystemVerilog. Tato rozhraní budou podrobně popsána v kapitole 2.2.

Informace o tomto produktu jsem čerpal z oficiálních webových stránek společnosti Mentor [3].

2.1.2 Riviera-PRO

Dalším z řady používaných RTL simulátorů je Riviera-PRO od společnosti Aldec. Podobně jako Questa Advanced Simulator je tento produkt určen pro zjednodušení a automatizaci verifikace. Podporovanými jazyky pro popis hardwaru jsou rovněž Verilog, SystemVerilog, VHDL a SystemC. Podporovaná rozhraní pro komunikaci se simulací jsou VPI pro jazyk Verilog, VHPI (angl. zk. *VHDL Procedural Interface*) pro jazyk VHDL a SystemVerilog DPI pro jazyk SystemVerilog.

Informace o tomto produktu jsem čerpal z oficiálních webových stránek společnosti Aldec [4].

2.2 Komunikační rozhraní

Pro komunikaci mezi simulací a externím uživatelským programem implementují RTL simulátory komunikační rozhraní. Tato rozhraní se mohou lišit jak použitým jazykem pro RTL popis obvodu, tak samotným RTL simulátorem. S jejich pomocí lze sledovat a kontrolovat simulaci pomocí externích programů. Nejčastěji jsou tato rozhraní implementovaná pro jazyky C a C++. Můžeme se ovšem setkat i s implementacemi pro jiné jazyky, například Python. Podrobněji zde budou zmíněny rozhraní VPI, FLI, VHPI a SystemVerilog DPI.

2.2.1 VPI

VPI je rozhraní pro přístup k simulaci modelu popsaného v jazyku Verilog pomocí jiných programovacích jazyků. Podpora pro toto rozhraní je implementována v obou zmíněných simulátorech. Podporu VPI lze přidat do simulace použitím parametru `-pli "<cesta_`

ke_knihovně_s_programem>" příkazu *vsim* používaného v obou simulátorech pro inicializaci simulace. Tato knihovna musí obsahovat pole ukazatelů na funkce `vlog_startup_routines`, ve kterém jsou uložny funkce, které se provedou po připojení knihovny do simulátoru. VPI je definováno v IEEE 1364 jako součást standardu PLI (angl. zk. *Programming Language Interface*). Pro použití VPI v programovacím jazyku C/C++ je nutné použít hlavičkový soubor *vpi_user.h*, který obsahuje veškeré definice funkcí VPI rozhraní. Rovněž musí vytvořený program obsahovat knihovnu s implementací tohoto rozhraní. Tato knihovna je odlišná pro použité RTL simulátory. Questa Advanced Simulator používá knihovnu *mtipli* a Riviera-PRO používá knihovnu *aldecpli*. Narozdíl od knihoven podléhá hlavičkový soubor pro VPI standardu a je pro všechny RTL simulátory stejný. Informace o tomto rozhraní jsem čerpal z knihy *The Verilog PLI Handbook* [16] a ze souboru *vpi_user.h*

VPI obsahuje tyto funkce:

- `vpiHandle vpi_register_cb (p_cb_data data);`

  Umožňuje vytvořit simulační callback, což je volání uživatelem definované funkce při určité události v simulaci. Mezi tyto události patří například start nebo konec simulace, změna hodnoty signálu, dosažení určitého simulačního času, atd. Konfigurace každého callbacku je uložena ve speciální struktuře, která je předána této funkci jako parametr, po jejím naplnění uživatelem. Funkce vrací odkaz na vytvořený callback, který je možné použít například k jeho následnému zrušení.

- `PLI_INT32 vpi_printf (PLI_BYTE8* format, ...);`

  Funguje podobně jako funkce `printf` v jazyku C, pouze místo vytisknutí textu na standardní výstup programu je text vytisknut na standardní simulační výstup a tudíž viditelný v RTL simulátoru a jeho záznamech. Podobně lze tisknout výstup do souboru pomocí rodiny funkcí `vpi_mcd`. Parametr `format` vyžaduje explicitní přetypování. Funkce vrací počet vytisknutých znaků nebo `EOF` při chybě.

- `vpiHandle vpi_handle_by_index (vpiHandle parent, PLI_INT32 index);`

  Aby mohli uživatelé přistupovat k jednotlivým signálům RTL designu, musí nejprve získat odkaz (v originále *handle*) na tyto signály. K tomu slouží ve VPI sada funkcí, které jsou schopny zjistit tyto odkazy na základě různých kritérií. Tato konkrétní funkce získává odkaz z nadřazeného objektu a indexu signálu v něm. Funkce vrací odkaz na nalezený signál nebo `NULL`, pokud nebyl nalezen.

- `vpiHandle vpi_handle_by_name (PLI_BYTE8* name, vpiHandle scope);`

  Obdoba funkce `vpi_handle_by_index`. Tato funkce získává odkaz pomocí názvu signálu popřípadě pomocí jeho relativní či absolutní cesty v hiearchii modelu. Parametr `name` vyžaduje explicitní přetypování. Parametr `scope` určuje počátek vyhledávání. Pokud je `NULL`, vyhledává se v celém modelu. Funkce vrací odkaz na nalezený signál nebo `NULL`, pokud nebyl nalezen.

- `void vpi_get_value (vpiHandle object, p_vpi_value value);`

  Slouží k přečtení aktuální hodnoty signálu. Odkaz na signál se přenáší parametrem `object`. Parametr `value` musí ukazovat na uživatelem alokovanou strukturu, do které se hodnota uloží. Pro klasický signál je jeho hodnota uložena v položce `integer` této struktury.

- `vpiHandle vpi_put_value (vpi_Handle object, p_vpi_value value, p_vpi_time time, PLI_INT32 flag);`

  Slouží k nastavení nové hodnoty signálu. Odkaz na signál se přenáší parametrem `object`. Nová hodnota je uložena ve struktuře, na kterou ukazuje parametr `value`, konkrétně v položce `integer`. Parametr `flag` určuje typ zpoždění. Pro naše účely nebude mít změna signálu zpoždění. Můžeme tedy použít konstantu `vpiNoDelay` a jelikož čas není potřeba znát, za parametr `time` můžeme dosadit `NULL`.

- `PLI_INT32 vpi_control (PLI_INT32 operation, ...);`

  Umožňuje ovládat některé aspekty simulace. Nejběžnějším použitím je ukončení simulace pomocí konstanty `VpiFinish` nebo pouze zastavení simulace pomocí konstanty `VpiStop`. Tyto konstanty se dosazují za parametr `operation`. U těchto operací je potřeba přidat ještě jeden celočíselný parametr kvůli úrovni diagnostických zpráv při ukončení. Přidaný parametr však nemá vliv na provedení operace.

Všechny zmíněné funkce pracující se signály jsou implementovány pro obecné objekty v jazyce Verilog. Pro tuto práci jsou ovšem relevantní pouze signály jako typy objektů a tudíž můžeme popis těchto funkcí zjednodušit, aby tomuto odpovídal. Ve VPI jsou k dispozici ještě další funkce, které ovšem pro účely této práce nejsou podstatné.

2.2.2 FLI

FLI je komunikační rozhraní pro jazyk VHDL, přes které lze připojit k simulaci externí programy. Toto rozhraní je primárně určeno pro jazyky C a C++ a není podporováno RTL simulátorem Riviera-PRO. Pro použití tohoto rozhraní v programu musíme ke zdrojovému kódu přidat hlavičkový soubor *mti.h*, ve kterém jsou připraveny veškeré potřebné definice (konstanty, funkce, atd.) pro jeho použití. Pro zpřístupnění rozhraní je potřeba při spuštění simulace přidat parametr `-foreign "<název_počáteční_funkce> <cesta_ke_knihovně_s_programem>"` příkazu *vsim*. Typ počáteční funkce je přesně definován na rozdíl od jejího jména, které si může uživatel specifikovat. Návratový typ této funkce musí být `void` a funkce musí mít čtyři parametry v tomto pořadí: `mtiRegionIdT, char*, mtiInterfaceListT*, mtiInterfaceListT*`. Pro tuto práci budou pravděpodobně tyto parametry nevyužity a tudíž není potřeba je podrobněji vysvětlovat. Výsledný program musí obsahovat knihovnu s implementací tohoto rozhraní. Pro Questa Advanced Simulator je to knihovna *mtipli*, stejná jako pro VPI rozhraní.

Informace o rozhraní FLI jsem čerpal z dokumentace k FLI [11] a ze souboru *mti.h*. Rozhraní obsahuje tyto funkce:

- `mtiProcessIdT mti_CreateProcess (char* name, mtiVoidFuncPtrT func, void* param);`

  Umožňuje vytvoření nového VHDL procesu (nikoliv procesu jako samostatně běžícího programu), který při své aktivaci provede uživatelem definovanou funkci, která je předána jako parametr `func`. Případný parametr této funkce je předán parametrem `param`. Parametr `name` určuje jméno procesu. Parametry `name` a `param` mohou být `NULL`, pokud je není třeba. Tato funkce proces pouze vytváří, nespecifikuje, kdy má být spuštěn. Návratovou hodnotou je identifikátor nového procesu.

- `void mti_ScheduleWakeup (mtiProcessIdT process_id, mtiDelayT delay);`

Určuje čas spuštění VHDL procesu od aktuálního simulačního času. Identifikátor procesu získaný funkcí `mti_CreateProcess` je předán jako parametr `process_id` a zpoždění od současného času v jednotkách simulačního času je předáno parametrem `delay`.

- `void mti_Sensitize (mtiProcessIdT process_id, mtiSignalIdT signal_id, mtiProcessTriggerT trigger);`

  Další možností jak specifikovat čas spuštění VHDL procesu je nastavit citlivost procesu na určitý signál pomocí této funkce. Takovýto proces se poté aktivuje při každé změně daného signálu. Identifikátor procesu i odkaz na signál jsou předány pomocí parametrů. Pomocí dalšího parametru lze změnit chování procesu na aktivaci při aktivní hladině signálu místo změny signálu.

- `void mti_AddLoadDoneCB (mtiVoidFuncPtrT func, void* param);`

  Umožňuje zaregistrování volání uživatelem definované funkce po úspěšném nahrání modelu před startem simulace. Parametr `param` určuje volitelný parametr dané funkce, při nevyužití může být `NULL`.

- `void mti_AddQuitCB (mtiVoidFuncPtrT func, void* param);`

  Obdoba funkce `mti_AddLoadDoneCB`. Umožňuje zaregistrování volání uživatelem definované funkce při ukončení simulace.

- `void mti_PrintMessage (char* message);`

  Vypisuje text předaný parametrem na standardní simulační výstup.

- `mti_PrintFormated (char* format, ...);`

  Vypisuje formátovaný text (podobně jako funkce `printf` v jazyku C) na standardní simulační výstup.

- `void mti_Break (); void mti_Quit (); void mti_FatalError ();`

  Tyto funkce umožňují zastavit simulaci (každá vlastním způsobem jak napovídají jejich názvy).

- `mtiSignalIdT mti_FindSignal (char* name);`

  Podobně jako ve VPI je důležité číst a nastavovat honoty jednotlivých signálů. Nejprve je třeba signál najít a získat na něj odkaz (v originále *signal ID*). Toho je nejsnadněji dosaženo touto funkcí, která vyhledá signál na základě jeho jména nebo absolutní či relativní cesty v modelu. Pokud signál není nalezen, funkce vrací `NULL`. K získání odkazu na signál jsou v FLI implementovány i další funkce, například `mti_FirstSignal` a `mti_NextSignal`.

- `mtiInt32T mti_GetSignalValue (mtiSignalIdT signal_id);`

  Umožňuje zjistit hodnotu signálu. Odkaz na signál (získaný například funkcí `mti_FindSignal`) je předán jako parametr `signal_id`. Tuto funkci lze použít kdykoliv v průběhu simulace a vrací vždy aktuální hodnotu signálu jako hodnotu výčtu. V tomto výčtu je logická 0 reprezentována hodnotou `2` a logická 1 hodnotou `3`.

- `void mti_SetSignalValue (mtiSignalIdT signal_id, long value);`

  Nastavuje novou hodnotu signálu. Nová hodnota signálu a odkaz na tento signál jsou předány parametry. Datový typ hodnoty se liší podle typu signálu. Obecně je tento parametr datového typu `long` pro skalární datové typy signálů. Hodnota parametru `value` je podobně jako v předchozí funkci pouze indexem do výčtu.

FLI má podstatně větší rozsah funkcí než základní VPI. Existují zde funkce pro přístup k proměnným jazyku VHDL, pro pokročilé řízení signálů, pro práci s regiony a podobně.

2.2.3 VHPI

VHPI není podporováno simulátorem Questa Advanced Simulator. Používá se pro propojení simulace modelu popsaného v jazyce VHDL s programovacím jazykem C/C++. Pro použití tohoto rozhraní je potřeba přidat do programu hlavičkové soubory *vhpi_user.h* a *aldecpli.h* pro Riviera-PRO simulátor. V souboru *vhpi_user.h* jsou deklarovány všechny funkce a konstanty tohoto rozhraní. Rovněž je třeba přidat k programu knihovnu s implementací tohoto rozhraní. Pro RTL simulátor Riviera-PRO se jedná o knihovnu *aldecpli*. Simulace poté musí být spuštěna se stejným parametrem jako v případě VPI. Funkce této knihovny jsou velice podobné funkcím z rozhraní VPI, proto jsou zde uvedeny i obdobné funkce z VPI pokud je to možné.

Informace o rozhraní VHPI jsem čerpal z článku *VHPI Aplications* [8].

- `vhpiHandleT vhpi_register_cb (vhpiCbDataT* cb_data_p, int32_t flags);`

  Obdoba funkce `vpi_register_cb`. Umožňuje vytvořit simulační callback. Parametr `cb_data_p` je struktura obsahující informace o callbacku a parametr `flags` přidává dodatečné vlastnosti této funkci. Návratová hodnota je `NULL`, pokud žádná dodatečná vlastnost není specifikována.

- `int vhpi_printf (const char* format, ...);`

  Obdoba funkce `vpi_printf`. Zprostředkovává tisk formátovaného textu na standardní simulační výstup. Na rozdíl od svého protějšku ve VPI není nutné přetypování parametru. Vrací počet vytisknutých znaků nebo `-1` při chybě.

- `vhpiHandleT vhpi_handle_by_name (const char* name, vhpiHandleT scope);`

  Obdoba funkce `vpi_handle_by_name`. Umožňuje získat referenci na signál na základě jeho jména nebo cesty pomocí parametru `name`. Parametr `scope` určuje objekt typu `region`, ve kterém se bude vyhledávat. Pokud je tento parametr `NULL`, vyhledává se v celé hiearchii modelu. Funkce vrací odkaz na nalezený signál nebo `NULL` v případě nenalezení.

- `int vhpi_get_value (vhpiHandleT expr, vhpiValueT* value_p);`

  Obdoba funkce `vpi_get_value`. Slouží k přečtení aktuální hodnoty signálu. Tato funkce se podstatně liší od svého protějšku ve VPI. V jazyku Verilog je hodnota signálu předávána jako číslo `0` nebo `1`. V jazyku VHDL je hodnota signálu předávána jako jeda hodnota z výčtu. Ve VHDL může signál nabývat i jiných hodnot než `0` nebo `1`. Existují zde i hodnoty reprezentující stavy jako například nenastaveno, neznámo, apod. Toto platí i pro rozhraní FLI, protože také pracuje s jazykem VHDL. Parametr `expr` je odkaz na čtený signál a parametr `value_p` je ukazatel na strukturu, do které

se načte hodnota signálu. Pro klasický signál je výsledná hodnota uložena v položce `intg` této struktury. Pro klasické signály funkce vrací `0` při úspěchu nebo zápornou hodnotu při chybě.

- `int vhpi_put_value (vhpiHandleT object, vhpiValueT* value_p, vhpiPutValueModeT mode);`

  Obdoba funkce `vpi_set_value`. Slouží k nastavení nové hodnoty signálu. Mezi touto funkcí a jejím protějškem ve VPI existuje stejný rozdíl jako u předchozí funkce. Parametr `object` určuje nastavovaný signál, parametr `value_p` obsahuje ukazatel na strukturu s nastavenou hodnotou signálu a parametr `mode` určuje mód změny. Existuje několik módů změny signálu, pro nás je ale nejpoužitelnější mód pod konstantou `vhpiDepositPropagate`, který změní hodnotu daného signálu a přepočítá všechny signály závislé na tomto signálu. Funkce vrací `0` pokud nenastala chyba, jinak nenulovou hodnotu.

- `int vhpi_control (vhpiSimControlT command);`

  Obdoba funkce `vpi_control`. Umožňuje ukončit simulaci. Povolené hodnoty parametru `command` jsou `vhpiStop` pro zastavení simulace a `vhpiFinish` pro ukončení simulace. Návratová hodnota je `0` při úspěchu příkazu, jinak `1`.

Rozhraní VHPI bylo vytvořeno s podobnými funkcemi a podobným principem jako rozhraní VPI pro jazyk Verilog. Liší se pouze vnitřní implementací a některými drobnými úpravami, ale jejich funkčnost je velice podobná.

2.2.4 SystemVerilog DPI

SystemVerilog DPI je rozhraní mezi jazykem SystemVerilog a jinými (cizími) jazyky jako je například C, C++ nebo Python. Toto rozhraní pracuje velice odlišně od předchozích. Rozhraní VPI, FLI a VHPI fungují na principu přímého napojení externího programu do simulace. SystemVerilog DPI umožňuje pouze kombinovat více jazyků při vytváření samotného modelu. Umožňuje napsání funkcí v jazyku SystemVerilog, které mohou být volány z programu v jiném jazyce a obráceně. V dříve uvedených rozhraních jsme se mohli napojit na simulaci bez jakéhokoliv zásahu do popisu modelu, nemuseli jsme žádným způsobem upravovat zdrojové kódy v jazycích Verilog nebo VHDL. Tato rozhraní lze sice použít podobným způsobem jako SystemVerilog DPI, ale je dobré vyhnout se konkrétním úpravám modelu kvůli možnosti globálního použití programu pro více modelů. Také jsme v dříve zmíněných rozhraních mohli ovlivňovat a kontrolovat simulaci pomocí předpřipravených funkcí. Tohoto nelze dosáhnout při použití SystemVerilog DPI, protože toto rozhraní neobsahuje žádné funkce pracující se simulací. Pokud bychom tedy chtěli zakomponovat toto rozhraní do simulace, musíme změnit zdrojový kód v jazyce SystemVerilog, aby volal uživatelem definované funkce v jiném programovacím jazyce.

Toto rozhraní je velice jednoduché na pochopení, protože nenabízí žádné speciální funkce nebo datové typy, ale veškerá komunikace je definována uživatelem pouze importováním, exportováním a voláním jednotlivých funkcí v obou směrech komunikace. Informace o tomto rozhraní jsem čerpal ze standartu *IEEE Standard for SystemVerilog* [12] a z článku *SystemVerilog DPI Tutorial* [5].

- Import do jazyka SystemVerilog se realizuje konstrukcí `import "DPI" function <hlavička_funkce>;`.

- Export z jazyka SystemVerilog se realizuje konstrukcí `export "DPI" function <název_funkce>;`.
- Import do jazyka C se realizuje pomocí konstrukce `extern <hlavička_funkce>;`.
- Export z jazyka C je automatický pro každou běžně definovanou funkci.

Zdrojový kód jazyka C musí obsahovat soubor *svdpi.h*. Zdrojové kódy v jazycích C i SystemVerilog musí být při načítání simulace přeloženy současně a musí být správně spojeny před zahájením simulace. Simulátory Questa Advanced Simulator a Riviera-PRO toto zajistí automaticky, pokud jsou tyto soubory oba načítány a přeloženy simulátory při načítání a překladu modelu.

SystemVerilog	C
byte	char
int	int
longint	long long
shortint	short int
real	double
shortreal	float
chandle	void*
string	char*

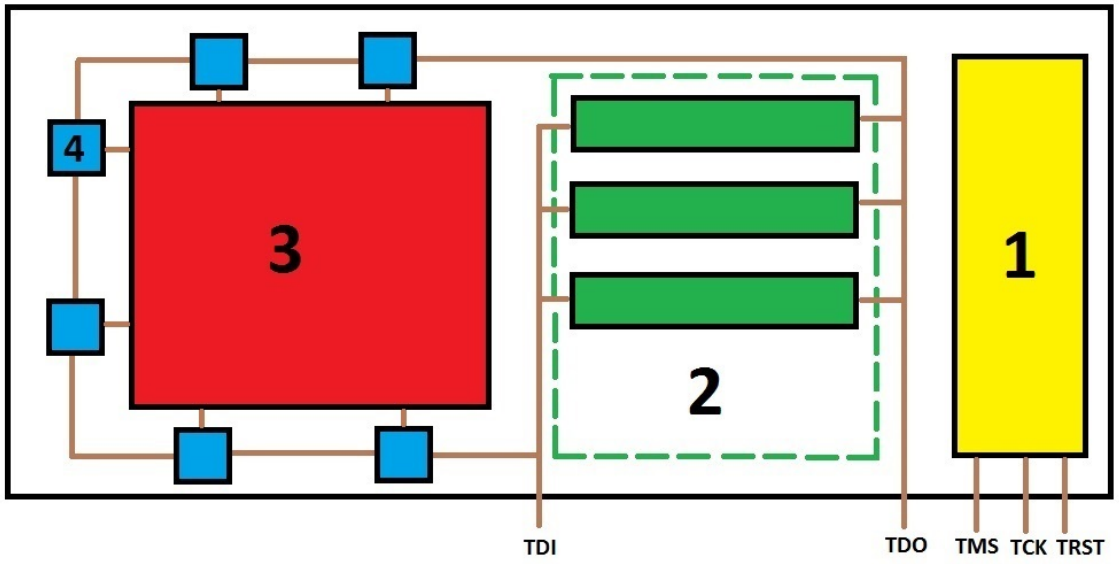
Obrázek 2.2: Mapování datových typů v DPI

Mohou nastat drobné komplikace u datových typů parametrů v exportovaných a importovaných funkcích a také v návratových datových typech funkcí. Proto je nutné vědět, jakým způsobem se na sebe mapují datové typy jazyků C a SystemVerilog. Mapování základních typů je zobrazeno v tabulce v obrázku 2.2.

Kapitola 3

JTAG a Nexus

JTAG je zkrácený název pro standard *IEEE 1149: Standart test access port and boundary scan architecture*. Základní schéma standardu JTAG se nachází na obrázku 3.1. Obecně slouží tento standard pro sjednocení a zjednodušení testovacích a ladících principů na hardwarových zařízeních.



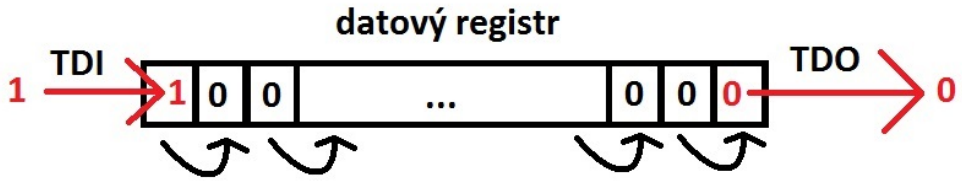
Obrázek 3.1: JTAG základní schéma

Na obrázku 3.1 je číslem 1 a žlutou bravou označený TAP kontrolér, číslem 2 a zelenou barvou označeny pomocné JTAG registry, číslem 3 a červenou barvou označeno jádro a funkční logika modelu, číslem 4 a modrou barvou jsou označeny *boundary scan cells* a hnědou barvou jsou označeny všechny datové vodiče.

Nexus je zkrácený název pro standard *IEEE-ISTO 5001*. Zabývá se podobně jako JTAG standard sjednocením testovacích a ladících technik do jednoho univerzálního rozhraní.

3.1 Funkčnost JTAG rozhraní

Základní princip JTAG rozhraní zpočívá vtom, že se ke vstupním a výstupním pinům zařízení připojí speciální registry, takzvané *boundary scan cells*. V ladícím režimu se uchovávají aktuální hodnoty vstupů nebo výstupů v těchto buňkách. Tyto registry lze poté číst nebo do nich zapisovat pomocí JTAG rozhraní a tímto lze ladit harware přímo za běhu. Jednotlivé buňky jsou sériově propojeny a tvoří jeden velký posuvný registr (*boundary scan register*, někdy též označován jako hlavní datový registr). Přístup k jednotlivým buňkám je realizován postupným posouváním registru a čtením hodnot, které jsou na konci registru a tudíž jsou posunuty ven z něho na výstupní port TDO. Při tomto posouvání lze rovněž nastavit nové hodnoty těchto registrů pomocí portu TDI. Hodnoty na tomto portu jsou při posunu registru posunuty na jeho začátek. Tímto způsobem je realizován přístup pro zápis i čtení u všech vstupních/výstupních pinů připojených k hlavnímu datovému registru. Postup čtení z hlavního datového registru a zápisu do něj je graficky znázorněn na obrázku 3.2.



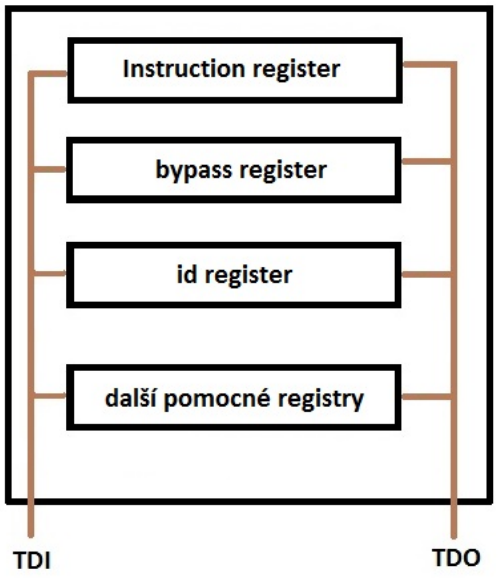
Obrázek 3.2: Princip posuvných registrů v JTAG rozhraní

JTAG standard dále popisuje registry, které jsou v zařízení nutné pro ovládání ladících funkcí. Čtení a zápis týkající se těchto a jiných nepovinně přítomných registrů je realizován podobným způsobem jeko přístup k hlavnímu datovému registru. Standardně se nachází v zařízení podporujícím JTAG minimálně tři pomocné registry:

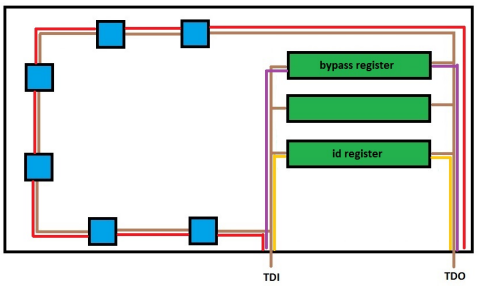
- `id register` (identifikační registr) obsahuje jednoznačnou identifikaci zařízení a je určen pouze pro čtení.
- `bypass register` (obcházecí registr) je jednobitový registr, který slouží pro propojení TDI a TDO portů. Pomocí tohoto registru lze obejít celý boundary scan posuvný registr a tím efektivně znepřístupnit toto zařízení pro ladění. Tento registr se využívá pouze za specifického stavu zařízení, kdy není třeba ho ladit a testovat.
- `instruction register` (instrukční registr) je poslední povinný registr. V tomto registru je uložena instrukce, kterou JTAG rozhraní právě provádí. Tento registr je určen pro čtení i pro zápis.

Všechny tyto registry jsou posuvné stejně jako boundary scan registr a lze pomocí čtení z nich a zápisu do nich ovládat veškeré ladění a testování zařízení.

Existuje několik instrukcí, které musí být povinně implementovány v každém zařízení podporujícím JTAG. Tyto instrukce jsou reprezentovány sekvencí binárních čísel, která může být odlišná na každém typu zařízení. Vykonávání instrukce započne po nahrání její bitové reprezentace do instrukčního registru. Následuje seznam těchto instrukcí.



Obrázek 3.3: Pomocné registry JTAG rozhraní



Obrázek 3.4: Datové cesty v JTAG rozhraní

- `BYPASS` instrukce propojuje porty TDI a TDO přes obcházecí registr a tímto způsobem učiní zařízení neladitelným, dokud není instrukce změněna. Toho se často využívá při sériovém zapojení více zařízení za sebou a obejití jednoho nebo více z nich. Datová cesta mezi porty TDI a TDO při vykonávání této instrukce je vyznačena fialovou barvou na obrázku 3.4.
- `EXTEST` instrukce připojuje porty TDI a TDO k hlavnímu datovému posuvnému registru a tudíž následně lze přečíst hodnoty jednotlivých vstupních a výstupních pinů na jádře zařízení. Datová cesta mezi porty TDI a TDO při vykonávání této instrukce je vyznačena červenou barvou na obrázku 3.4.
- `SAMPLE (PRELOAD)` instrukce funguje podobně jako instrukce EXTEST, pouze s tím rozdílem, že ponechává zařízení v běžném chodu a ne v ladícím módu. Často se používá pro nastavení boundary scan registru před samotným testováním.

- `IDCODE` instrukce nemusí být podle standardu povinně implementována v každém zařízení. Propojuje porty TDI a TDO s identifikačním registrem, jak je nakresleno žlutou barvou na obrázku 3.4. Tento registr by měl být přístupný pouze pro čtení.

Existují i další volitelné instrukce podporované určitými zařízeními pro lepší přístup k datům a k usnadnění ladění.

Celá řídící logika JTAG rozhraní je řízena pomocí speciálního zařízení, které se nazývá TAP kontrolér. Jednotlivá zařízení podporující JTAG lze libovolně řetězit za sebou v boundary scan registru. Každé takovéto zařízení obsahuje vlastní JTAG registry i TAP kontrolér umožňující ovládání těchto registrů. Propojení mezi zařízeními tvoří pouze signály TDI a TDO. Tímto lze dosáhnout ladění více zařízení najednou, protože hlavní posuvný registr obsahuje data všech zařízení. Pro vybrání a ovládání konkrétního zařízení, bez možnosti zásahu do dalších, se využívá právě instrukce BYPASS, která dokáže odstínit jednotlivé vstupní a výstupní piny zařízení od JTAG signálů.

Informace o JTAG rozhraní jsem čerpal částečně ze článku *What is JTAG and how can I make use of it?* [9] a především ze článku *Technical Guide to JTAG* [6].

3.2 TAP kontrolér

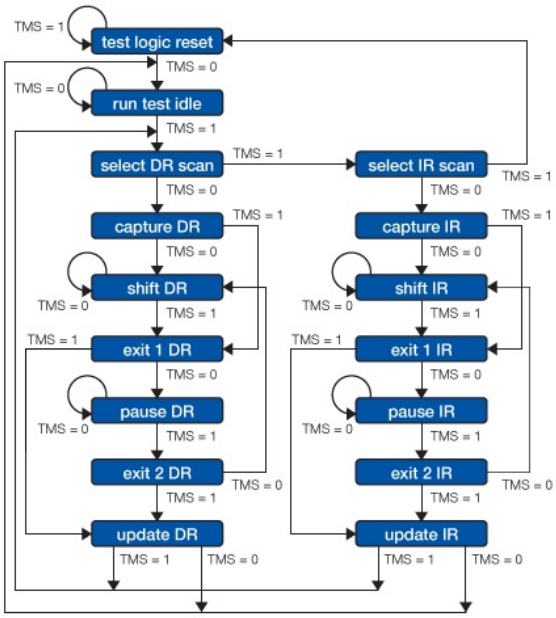
Test Access Port Controller je ve své podstatě stavový automat, který řídí logiku celého JTAG zařízení. Do TAP kontroléru jsou přivedeny všechny řídící signály JTAG standardu. Jedná se o tři signály: TCK, TMS a TRST. Signály TDI a TDO jsou přivedeny přímo k registrům zařízení a tudíž nejsou zpracovávány v této kapitole.

- Signál TCK je hodinový signál pro TAP kontrolér. Vždy na nástupné hraně tohoto signálu jsou vzorkovány nové hodnoty signálů TMS a TDI a jsou dále zpracovávány. Na sestupné hraně tohoto signálu je vzorkován signál TDO, který by měl být validní až do další nástupné hrany signálu TCK. Frekvence signálu TCK by měla být alespoň třikrát větší, než frekvence signálu CLK na daném zařízení, jinak je možné, že zařízení nebude stíhat zpracovávat data přes JTAG rozhraní a dojde k nespecifikovaným stavům.

- Signál TRST je podle standardu volitelný. Tento signál slouží pro reset TAP kontroléru do původního stavu. Reset se děje při sestupné hraně tohoto signálu.

- Signál TMS ovládá vnitřní stavový automat. Každý stav tohoto automatu má dva přechody, které se aktivují podle hodnoty signálu TMS při jeho vzorkování na vzestupné hraně TCK.

Přesný popis stavového automatu je k dispozici na obrázku 3.5: TAP stavový automat. Automat se skládá celkem ze 16 stavů, které jsou vyjmenovány níže.



Obrázek 3.5: TAP stavový automat
[6]

- Stav *test logic reset* je výchozí stav automatu. A také je to stav, do kterého se automat dostane při sestupné hraně signálu TRST. Automat je sestaven tak, že z libovolného stavu se lze do tohoto stavu dostat pomocí pěti cyklů signálu TCK s TMS signálem nastaveným na logickou 1.
- Stav *run test idle* je běžným stavem automatu, pokud se neprovádí žádná akce v JTAG rozhraní.
- Stavy *select DR scan* a *select IR scan* vybírají mezi instrukčním a datovým registrem. Všechny následující stavy jsou k dispozici pro IR i pro DR.
- Stavy *capture DR* a *capture IR* zachytí aktuální hodnotu zvoleného registru a zpřístupní ji dalším operacím.
- Stavy *shift DR* a *shift IR* posouvají vybraný registr. Na první pozici registru se zapíše hodnota signálu TDI a poslední bit registru je zpřístupněn jako nová hodnota signálu TDO.
- Stavy *exit 1 DR*, *exit 2 DR*, *exit 1 IR* a *exit 2 IR* slouží pro potvrzení úprav daného registru před jeho uložením a pro případné vrácení se k dalším změnám nebo pozastavení před potvrzením.
- Stavy *pause DR* a *pause IR* slouží pro pozastavení automatu mezi dokončením úprav a jejich uložením.
- Stavy *update DR* a *update IR* zapisují novou hodnotu zvoleného registru přístupného přes signály TDI a TDO do vnitřního registru zařízení, kde se hodnoty aplikují.

Mezi všemi těmito stavy se lze přepínat pomocí nastavení signálu TMS při tiku hodin TAP kontroléru.

3.3 Nexus

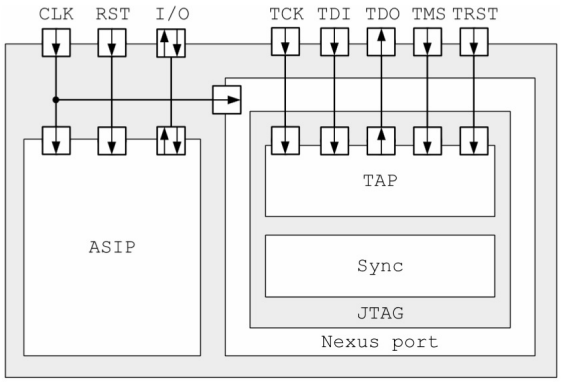
Nexus standard je dalším ladícím a testovacím standardem podobně jako JTAG. Nexus je oproti JTAG standardu pokročilejší a komplexnější. Existují čtyři třídy Nexus rozhraní. Přičemž první třída využívá JTAG rozhraní pro přenos dat. Vyšší třídy používají Nexus Auxiliary Port rozhraní. Nexus standard specifikuje širokou škálu registrů, které je doporučeno implementovat pro správnou funkcionalitu níže uvedených operací. Nechybí zde ani podpora breakpointů a watchpointů. Breakpoint je místo v programu, na kterém se má pozastavit jeho vykonávání. Často se jedná o řádek programu nebo instrukci. Watchpoint funguje podobně jako breakpoint, ale není vázán na konkrétní místo programu. Místo toho je vázán na vyhodnocení zadaného výrazu. Nexus rozhraní by mělo umožňovat níže uvedené operace.

- čtení z a zápis do uživatelských registrů v ladícím režimu
- čtení z a zápis do uživatelské paměti v ladícím režimu
- vstup do ladícího režimu z běžného režimu nebo z resetu
- opuštění ladícího režimu
- zastavení provádění na breakpointu a následné přepnutí do ladího režimu
- čtení identifikace zařízení
- upozornění na dosažení watchpointu
- nastavení breakpointů a watchpointů
- krokování programu po instrukcích

Technické detaily Nexus standardu jsou příliš obsáhlé a pro účely této práce téměř nepodstatné, proto zde nebudou popsány. Informace o Nexus rozhraní jsem čerpal ze článku *Freescale Nexus 5001 Software Debug Interfaces* [1]

3.4 JTAG adaptér

JTAG adaptér je program a knihovna od společnosti Codasip. Umožňuje komunikaci mezi Nexus rozhraním a externím programem (často simulátorem nebo debuggerem) pomocí JTAG rozhraní. Hardwarový model, který je možné použít s JTAG adaptérem by měl vypadat podle obrázku 3.6.



Obrázek 3.6: Model hardwaru pro práci s JTAG adaptérem
[13]

JTAG adaptér umožňuje implementaci pluginů, které musí splňovat předem zadané rozhraní. Tyto pluginy slouží jako prostředníci mezi daty posílanými z JTAG adaptéru a různými metodami přístupu k reálnému hardwaru nebo jeho simulaci. Pro registraci pluginu JTAG adaptéru je nutné definovat strukturu obsahující veškeré informace o daném pluginu, které JTAG adaptér potřebuje. Ukazatele na funkce v této struktuře budou podrobněji popsány níže. Struktura obsahuje položky zobrazené v obrázku 3.7.

```
unsigned m_Version; // verze rozhraní jtag adaptéru, kterou plugin používá
const char* m_Name; // unikátní jméno pluginu
void* (*m_CreateFnc)();
void (*m_DestroyFnc)(void* userData);
bool (*m_InitFnc)(void* userData, const CodasipPluginConfig* config);
bool (*m_SendRecvFnc)(void* userData, BYTE* data, const unsigned bitsize);
const char* (*m_ErrorFnc)(void* userData);
```

Obrázek 3.7: Položky struktury s informacemi o RTL pluginu

Pro použití JTAG adaptéru s reálným hardwarem se používá spustitelný program JTAG adaptéru, který po zapnutí vytvoří specifikovaný plugin a zahájí pomocí něho komunikaci s Nexus rozhraním na zařízení. Při použití s hardwarovým simulátorem tento postup použít nelze, protože ve většině případů musí být nejprve spuštěn simulátor a až v něm je nastaveno, že se budou přijímat příkazy z externího programu. Proto je nutné použít JTAG adaptér jako knihovnu, která se bude nahrávat při spuštění simulátoru.

Následuje podrobný popis jednotlivých funkcí struktury obsahující informace o pluginu společně s funkcí `Head`, která slouží pro inicializaci a spuštění JTAG adaptéru.

- `void Head (std::vector<std::string> args,`
  `void (*TerminateProgram)(const char* message));`

  Funkce `Head` je jediným vstupním bodem knihovny JTAG adaptéru. Jejími parametry jsou `args`, obsahující parametry pro JTAG adaptér (stejné jako u spustitelného programu), a `TerminateProgram`, což je ukazatel na funkci, která zajišťuje korektní ukončení simulace a programu v daném pluginu. Knihovna JTAG adaptéru také zachytává výjimky datového typu `int`, které může vygenerovat plugin. Při zachycení

takovéto výjimky je program ukončen pomocí funkce `TerminateProgram` s odpovídající chybovou hláškou.

- `void* (*m_CreateFnc)();`

  Tato funkce slouží pro vytvoření nové instance požadovaného pluginu. Volá se při inicializaci JTAG adaptéru a vrací obecný ukazatel. Tento ukazatel je poslán jako parametr do každé další funkce pluginu a lze pomocí něj přenášet data specifická pro tento plugin mezi jednotlivými funkcemi.

- `void (*m_DestroyFnc)(void* userData);`

  Tato funkce slouží pro korektní ukončení pluginu a uvolnění paměti. Funkce nevrací nic a jejím parametrem je ukazatel vrácený z funkce `m_CreateFnc`. Tato funkce se volá z JTAG adaptéru vždy před ukončením jeho činnosti. Tedy pokud uživatel ukončí program nebo při chybě. Je garantováno, že po zavolání této funkce už nebudou volány žádné další funkce z této struktury.

- `bool (*m_InitFnc)(void* userData, const CodasipPluginConfig* config);`

  Tato funkce je volána vždy po vytvoření pluginu a zajišťujě jeho inicializaci. Návratovou hodnotou je `true` při úspěšné inicializaci, jinak `false`. Pokud inicializace neproběhne úspěšně, je zavolána funkce `m_DestroyFnc`. Prvním parametrem této funkce je ukazatel vrácený z funkce `m_CreateFnc`. Druhým parametrem je ukazatel na speciální strukturu obsahující všechny údaje potřebné ke konfiguraci pluginu. V této struktuře se nachází především cesta ke konfiguračnímu souboru pro daný plugin, který bývá většinou specifický pro konkrétní RTL popis. Další důležitou částí je seznam argumentů, které byly předány JTAG adaptéru po jeho spuštění a nebyly jím rozpoznány, tudíž se předpokládá, že mají specifický význam pro použitý plugin.

- ```
  bool (*m_SendRecvFnc)
      (void* userData, BYTE* data, const unsigned bitsize);
  ```

  Tato funkce je nejčastěji volaná funkce pluginu. Je volána pouze po úspěšné inicializaci a zajišťuje veškerou další komunikaci mezi JTAG adaptérem a pluginem. Pokaždé, když JTAG adaptér potřebuje komunikovat se simulací, zavolá tuto funkci a v parametru `data` pošle bity, které plugin přepošle do Nexus rozhraní simulovaného modelu pomocí JTAG rozhraní. Za každý takto poslaný bit je přes JTAG rozhraní vrácen jeden bit s odpovědí pro JTAG adaptér. Podrobně je tento princip popsán na obrázku 3.2. Po odeslání všech bitů je tedy k dispozici odpověď stejně dlouhá, jako byl požadavek. Tato odpověď je poslána zpět JTAG adaptéru ve stejném parametru, jako přišel požadavek. Parametr `bitsize` udává počet bitů požadavku/odpovědi. Parametr `userData` je ukazatel vrácený z funkce `m_CreateFnc`. Funkce vrací `true` při úspěšném zpracování požadavku, jinak `false`. Pokud komunikace neproběhne úspěšně, je zavolána funkce `m_DestroyFnc` a program je ukončen s chybou.

- `const char* (*m_ErrorFnc)(void* userData);`

  Tato funkce slouží pro zjištění typu chyby uvnitř pluginu. Typicky se volá před zničením pluginu, pokud některá z dříve provedených funkcí pluginu proběhla neúspěšně. Jediným parametrem je ukazatel vrácený z funkce `m_CreateFnc`, ovšem pokud selže vytvoření pluginu, je hodnotou tohoto parametru `NULL`. Návratovou hodnotou by měla být poslední zaznamenaná chyba uvnitř pluginu.
  ```

Funkčnost JTAG adaptéru se dá ovlivňovat jeho argumenty, které jsou v případě knihovní verze předávány pomocí konfiguračního souboru. Mezi tyto argumenty patří:

- `-p <port>` - argument udávající číslo portu, přes který je možno se připojit na JTAG adaptér pomocí debuggeru.
- `--plugin <plugin>` - argument udávající název pluginu, který má být JTAG adaptérem použit.
- `-C <konfigurace_nexus>` - argument obsahující cestu k soubrou s informacemi o modelu a jeho Nexus rozhraní.
- `-c <konfigurace_plugin>` - argument obsahující cestu k soubrou se specifickými informacemi o modelu pro konkrétní plugin.

Informace o JTAG adaptéru jsem získal od společnosti Codasip, především z dokumentů *Codasip Studio Technical Reference Manual* [13] a *Codasip Studio User Guide* [14] a také od zaměstnanců této společnosti.

Kapitola 4

Návrh a implementace

4.1 Hlavní návrh

Výsledkem mé implementace je plugin pro knihovnu obsahující JTAG adaptér, který jsem nazval *RTL plugin*. Tento plugin je součástí knihovny JTAG adaptéru. První věcí, kterou je třeba udělat, je implementovat rozhraní pro JTAG adaptér plugin, aby bylo možné jej do JTAG adaptéru zapojit. Toto rozhraní je popsáno v kapitole 3.4. Při inicializaci pluginu je potřeba správně nastavit propojení JTAG a Nexus rozhraní a také zpracovat případné parametry určené pro plugin. Všechny posílané informace jsou poté rozděleny na jednotlivé bity a ty jsou dále zpracovány. Kvůli možné implementaci více komunikačních rozhraní jsem se rozhodl implementovat pluginový systém i pro RTL plugin. Jednotlivé podpluginy by měly zajišťovat zpracování jednotlivých bitů informací v simulaci. RTL plugin by tedy neměl být závislý na konkrétních komunikačních rozhraních mezi simulací a externími programy. Tyto podpluginy by měly pravidelně kontrolovat dostupnost nových dat z RTL pluginu a popřípadě zastavit tento plugin, dokud nebudou požadované instrukce provedeny v simulaci. Pro účely této práce jsem implementoval pouze jediný podplugin, který se jmenuje *Questa plugin* a implementuje komunikační rozhraní VPI pro Questa Advanced Simulator. Questa plugin již není součástí JTAG adaptéru, ale tvoří samostatnou knihovnu, která ovšem linkuje knihovnu JTAG adaptéru dynamicky za běhu.

4.2 Spuštění simulace

První věcí, kterou jsem implementoval, bylo samotné připojení JTAG adaptéru k simulaci. Protože připojení knihovny do simulace je závislé na specifickém komunikačním rozhraní, je potřeba zajistit vstupní bod v knihovně Questa pluginu. Jak je zmíněno v kapitole 2.2.1, při linkování knihovny k simulaci jsou provedeny funkce, jejichž ukazatele jsou uloženy v poli `vlog_startup_routines`. Toto pole musí být exportováno z knihovny, musí být ukončené hodnotou `NULL` a musí být kompatibilní s jazykem C. Kvůli poslední podmínce je potřeba použít pro toto pole příkaz `extern "C"`. Všechny funkce, na které jsou v tomto poli ukazatele, musí být bez parametrů a musí mít návratový typ `void`. Implementace tohoto pole obsahuje pouze jedinou funkci `Inicializer`.

- `void Initializer ();`

  Tato funkce má dva primární úkoly.

  - spuštění a inicializace RTL pluginu a JTAG adaptéru

– zaregistrování funkce, která bude provolána při zahájení simulace

První úkol je zajištěn zavoláním funkce `Start`, která je vyexportována z RTL pluginu. Tato funkce bere za parametr strukturu s informacemi o daném pluginu. Tuto strukturu jsem vytvořil podle předlohy struktury z JTAG adaptéru, která identifikuje jeho pluginy. Tato funkce vrací hodnotu typu `bool`, která říká, zda byla inicializace dokončena úspěšně. Pokud se tato funkce nezdařila, není třeba provádět druhý úkol ani jakkoliv uvolňovat zdroje. Pouze je třeba vypsat odpovídající chybovou hlášku a simulaci ukončit. Bližší specifikaci pluginového systému a funkci `Start` je věnována kapitola 4.3.

Druhý úkol je poměrně jednoduchý na implementaci. Stačí vytvořit strukturu `s_cb_data` a vyplnit její položku `reason` konstantou `cbStartOfSimulation` a položku `cb_rtn` funkcí, která má být zavolána. V našem případě se jedná funkci `SimulationStarter`. Poté je třeba zaregistrovat tento callback pomocí funkce `vpi_register_cb`. Kvůli správnému uvolnění paměti je doporučeno po registraci zavolat funkci `vpi_free_object` s parametrem, který vrátila předchozí funkce.

4.3 Pluginový systém

Pluginový systém umožňující implementaci různých druhů komunikace se simulací jsem vytvořil podle vzoru pluginového systému JTAG adaptéru. Podobně jako pro pluginy JTAG adaptéru je i pro mé pluginy definována struktura, která uchovává důležité informace o jejich implementaci. Tato struktura obsahuje položky zobrazené v obrázku 4.1.

```
unsigned m_Version; // verze rozhraní RTL pluginu, kterou plugin používá
bool (*m_TickFnc)(void* userData, const bool tms, const bool tdi);
void (*m_DestroyFunc)
    (void* userData, const char* message, const bool simExit);
void* (*m_CreateFunc)();
bool (*m_FinishSimEventFunc)(const bool success);
bool (*m_IsSimEventFunc)();
bool (*m_SimInitFunc)(const bool success);
void (*m_WaitForDebuggerFunc)();
void (*m_SimEventFunc)()
const char* (*m_ErrorFunc)();
```

Obrázek 4.1: Položky struktury s informacemi o pluginu

Tato struktura musí být vytvořena a správně nainicializována před zavoláním funkce `Start` z RTL pluginu, které se tato struktura předává jako parametr. Každý plugin musí inicializovat první čtyři položky této struktury. První položkou je verze pluginu. Tato položka zabraňuje spuštění nekompatibilního pluginu, což je ošetřeno uvnitř RTL pluginu. Další tři položky jsou ukazatele na funkce, které plugin musí implementovat a které jsou provolávány z RTL pluginu. Zbytek struktury může zůstat neinicializován. Všechny zbývající položky jsou do struktury doplněny uvnitř funkce `Start`, kde RTL plugin poskytne ukazatele na vlastní funkce zajišťující komunikaci mezi ním a jeho pluginy. Po návratu z této funkce je tedy struktura plně inicializována a plugin může používat všechny její položky. Toto platí, i když návratová hodnota funkce je `false`, tudíž se nezdařila inicializace RTL pluginu.

V tomto případě však není doporučeno obsah struktury používat, protože by toto chování mohlo vést k neočekávaným stavům.

Následuje podrobný popis jednotlivých funkcí, jejichž ukazatele jsou umístěny ve výše zmíněné struktuře.

- `void* (*m_CreateFunc)();`

  Tato funkce slouží pro vytvoření nové instance požadovaného pluginu. Volá se při inicializaci RTL pluginu uvnitř funkce `Start` a vrací obecný ukazatel. Tento ukazatel je poslán jako parametr do každé další funkce pluginu a lze pomocí něj přenášet data specifická pro tento plugin mezi jednotlivými funkcemi.

- `void (*m_DestroyFunc)`
  `(void* userData, const char* message, const bool simExit);`

  Tato funkce slouží pro korektní ukončení pluginu a uvolnění paměti. Funkce nevrací nic. Jejím prvním parametrem je ukazatel vrácený z funkce `m_CreateFnc`. Tato funkce se volá z RTL pluginu vždy před ukončením jeho činnosti, ale pouze pokud byl plugin vytvořen. Tedy pokud uživatel ukončí program nebo při chybě vzniklé po inicializaci RTL pluginu. Je garantováno, že po zavolání této funkce už nebudou volány žádné další funkce z daného pluginu.

- `bool (*m_TickFnc)(void* userData, const bool tms, const bool tdi);`

  Tato funkce slouží pro zpracování jednoho tiku signálu TCK z JTAG rozhraní. Jejím prvním parametrem je ukazatel vrácený z funkce `m_CreateFnc`. Další dva parametry určují nové hodnoty signálů TMS a TDI, které má simulace nastavit. Tyto hodnoty se přenáší pomocí datavého typu `bool`, kde hodnota `false` znamená logickou 0 a hodnota `true` znamená logickou 1. I když simulace podporuje většinou i jiné hodnoty signálů (např. neznámo nebo nenastaveno), nastavovat lze pouze tyto dvě. Návratová hodnota této funkce je `true`, pokud simulace úspěšně nastavila signály a provedla tik, nebo `false`, pokud nastala chyba při vykonávání některé z podoperací simulace. V případě neúspěchu funkce je RTL plugin i JTAG adaptér s tímto obeznámen a simulace i program jsou ukončeny náležitým způsobem.

- `bool (*m_SimInitFunc)(const bool success);`

  Tuto funkci volá plugin ve chvíli, kdy je dokončena příprava simulace a plugin je připraven přijímat požadavky od RTL pluginu (pomocí funkce `m_TickFunc`). Dokud tato funkce nebude zavolána, bude RTL plugin čekat před zavoláním `m_TickFunc` funkce. Tento mechanismus jsem musel zavést kvůli případům, kdy inicializace RTL pluginu a JTAG adaptéru trvá kratší dobu než inicializace simulace. V daných případech se synchronizace mezi simulačním a řídícím vláknem vymyká kontrole a může způsobit neočekávané stavy systému. Tuto funkci je nutné zavolat i při neúspěšné inicializaci simulace. Pro odlišení úspěšné a neúspěšné inicializace slouží parametr `success`. Nutnost volání této funkce i při neúspěšné inicializaci je způsobena nutností synchronizovat dvě programová vlákna a obě dvě korektně ukončovat před samotným koncem programu. Podobný koncept je opakován i u některých dalších funkcí z tohoto seznamu. Synchronizaci vláken se bude podrobněji věnovat kapitola 4.4.

- `void (*m_SimEventFunc)()`

  Zavoláním této funkce dává plugin najevo, že přijal data z RTL pluginu a že je může začít simulační vlákno zpracovávat. Zpravidla by se měla volat uvnitř funkce `m_`

`TickFunc`. Zavolání této funkce způsobí, že funkce `m_IsSimEventFunc` bude vracet hodnotu `true`. Primárním účelem této funkce je synchronizace mezi simulačním a řídícím vláknem programu.

- `bool (*m_FinishSimEventFunc)(const bool success);`

  Zavoláním této funkce dává plugin najevo, že zpracoval přijatá data z RTL pluginu a že je připraven zpracovat další požadavek. Zavolání této funkce způsobí, že funkce `m_IsSimEventFunc` bude vracet hodnotu `false`. Tato funkce by nikdy neměla být volána bez předchozího zavolání funkce `m_SimEventFunc`. Primárním účelem této funkce je synchronizace mezi simulačním a řídícím vláknem programu. Podobně jako `m_SimInitFunc` má i tato funkce parametr `success`, kterým je dáno RTL pluginu najevo, že nastala chyba při vykonávání požadavku v simulačním vlákně. Tato funkcionalita je stejná jako ve funkci `m_SimInitFunc`.

- `bool (*m_IsSimEventFunc)();`

  Tato funkce slouží pluginu pro zjištění, zda právě probíhá zpracování požadavku z RTL pluginu. Výsledek této funkce závisí na stavu pluginu. Pokud byla zavolána funkce `m_SimEventFunc` a ještě po ní nebyla zavolána funkce `m_FinishSimEventFunc`, vrací tato funkce hodnotu `true`. V ostatních případech vrací hodnotu `false`. Primárně se tato funkce používá ke zjištění, zda má simulační vlákno povolení upravovat hodnoty v simulaci.

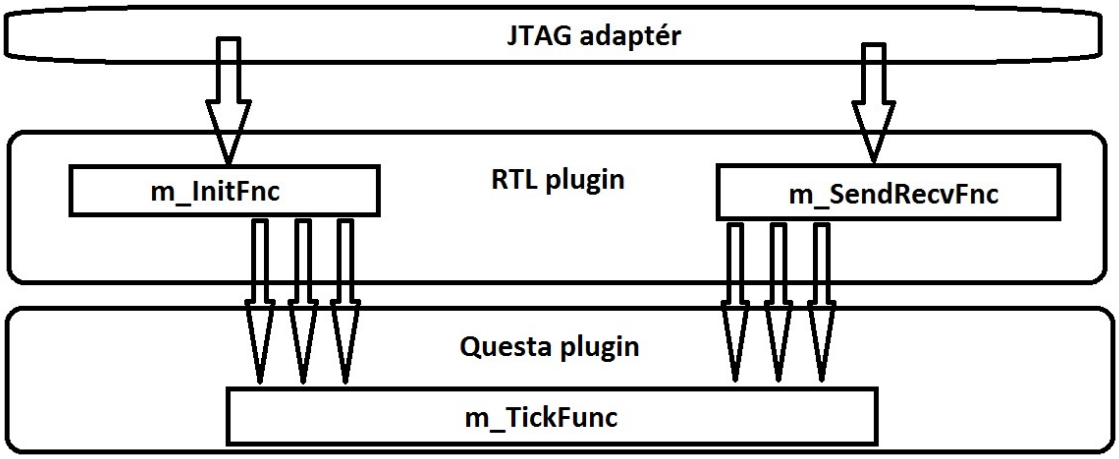
- `void (*m_WaitForDebuggerFunc)();`

  Tato funkce slouží výhradně ke korektnímu ukončení programu a obou běžících vláken. Po zavolání z pluginu zastaví vykonávání řídícího vlákna a počká na jeho korektní ukončení. Pokud běží obě vlákna, je toto jediná validní možnost, jak korektně ukončit program. Jinými metodami by mohlo dojít k neuvolnění alokovaných zdrojů a podobným problémům. V případě, že by řídící vlákno přestalo posílat požadavky do simulace a simulace by narazila na chybu, snažila by se tuto funkci použít k ukončení programu. Tato funkce ovšem bude čekat na ukončení řídícího vlákna, které ale dostane zprávu o chybě až při provádění dalšího příkazu. Tudíž je v této nepravděpodobné situaci nutné, aby uživatel zaslal JTAG adaptéru další příkaz, aby detekoval chybu a korektně se ukončil.

- `const char* (*m_ErrorFunc)();`

  Tato funkce má především informativní charakter. Funkci lze použít pro výpis chybových hlášek na simulační výstupy. Pokaždé, když nastane chyba v RTL pluginu nebo v JTAG adaptéru, je uložena nová hodnota poslední chyby. Tato hodnota se nastavuje například při zavolání funkce `m_SimInitFunc` nebo funkce `m_FinishSimEventFunc` s hodnotou parametru `false`. Tuto hodnotu vrací zmíněná funkce jako návratovou hodnotu datového typu `const char*`. Teoreticky se tato funkce dá využít i pro kontrolu chyby, která nastala v řídícím vlákně, ze simulačního vlákna. V implementaci Questa pluginu jsem ovšem zvolil jiný způsob kontroly této situace.

Grafické znázornění datových toků a provolávání funkcí oběma pluginovými systémy (předpřipravený pluginový systém JTAG adaptéru a pluginový systém RTL pluginu, který jsem implementoval) je vidět na obrázku 4.2.



Obrázek 4.2: Příklad provolávání funkcí přes pluginové systémy

4.4 Synchronizace vláken

Jak bylo nastíněno v předchozí kapitole, běh programu se skládá ze dvou nezávislých vláken. Vlákno, které používá komunikační rozhraní pro práci se simulací, jsem nazval *simulační vlákno*. Vlákno, které zpracovává příkazy a připravuje data na změnu simulace, jsem nazval *řídící vlákno*. Při nahrání pluginu ze simulátoru je spuštěno pouze jedno vlákno, a to simulační. Řídící vlákno je vytvořeno ke konci funkce `Start` uvnitř RTL pluginu. Od té doby je tedy třeba zajišťovat vzájemnou komunikaci těchto vláken. Většina synchronizace mezi těmito vlákny se děje pomocí funkcí popsaných v kapitole 4.3.

4.4.1 Použité prostředky

Pro implementaci vláken a jejich synchronizace jsem použil implementace z hlavičkového souboru *windows.h*. Pro jednoduchou synchronizaci (např. v `m_SimInitFunc`) jsou použity mutex (angl. *mutual exclusion*) objekty. Pro pokročilejší synchronizaci jsou použité podmínkové proměnné (angl. *condition variable*) a SRW (angl. *Slim Read/Write*) zámky. Nevýhodou takovéto implementace je nepřenositelnost programu na linuxové platformy. Výhodou je kompatibilita se staršími verzemi MinGW překladače, které nedokáží zpracovat standardní C++ implementace (například v hlavičkovém souboru *mutex*).

V implementaci jsou použity tyto datové typy definované v souboru *windows.h*:

- `HANDLE` pro vlákno a mutex. Tento datový typ obsahuje referenci na některý ze základních objektů definovaných ve *windows.h*. Mutex je základní synchronizační objekt, který může být zamknut a odemknut určitým vláknem. Pokud je zamknutý, ostatní vlákna, která ho chtějí zamknout, musí počkat než bude odemčen.
- `CONDITION_VARIABLE` pro podmínkovou proměnnou, což je pokročilý synchronizační objekt, který umožňuje podmínit čekání na uvolnění mutexu uživatelem definovanou podmínkou.

- `SRWLOCK` pro SRW zámek. Tento speciální typ mutexu umožňuje být zamknutý několika způsoby a používá se v kombinaci s podmínkovou proměnnou.

V implementaci jsou využity tyto funkce definované v souboru *windows.h*:

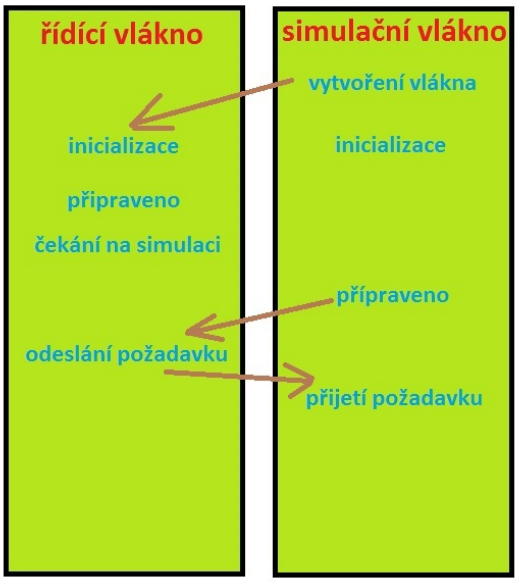
- `CreateThread` umožňuje vytvoření a spuštění nového vlákna v programu. Funkce, kterou má vlákno začít, a její volitelný argument jsou předány pomocí parametrů. Odkaz na nově vytvořené vlákno je vrácen jako návratová hodnota.
- `CreateMutex` umožňuje vytvoření a inicializaci nového mutexu. Odkaz na nově vytvořený mutex je vrácen jako návratová hodnota.
- `ReleaseMutex` slouží k odemknutí již zamčeného mutexu. Daný mutex je předán parametrem.
- `WaitForSingleObject` je funkce, kterou lze použít s více objekty. Pokud je jí předán parametrem objekt vlákna, volající vlákno čeká na ukončení tohoto vlákna. Pokud je parametrem mutex, uzamkne tento mutex, popřípadě před uzamčením čeká, než bude odemčen.
- `InitializeConditionVariable` inicializuje podmínkovou proměnnou před jejím prvním použitím. Daná podmínková proměnná je předána parametrem.
- `SleepConditionVariableSRW` uspí volající vlákno. To čeká, dokud nebude poslána notifikace pro podmínkovou proměnnou danou parametrem. Dalším parametrem je SRW zámek, který slouží jako mutex pro uživatelem definovanou podmínku použitou pro znovurozběhnutí vlákna.
- `WakeConditionVariable` pošle notifikaci pro podmínkovou proměnnou danou parametrem a tím znovurozběhne vlákno, které na tuto notifikaci čeká (pouze, pokud je splněna podmínka).

Některé ze zmíněných synchronizačních mechanismů jsou dostupné pouze od verze Windows Vista. Pokud je tedy chceme využít, musíme definovat speciální konstanty před použitím souboru *windows.h*, kterými říkáme programu, že může zpřístupnit tyto implementace závislé na verzi systému Windows.

4.4.2 Synchronizace v praxi

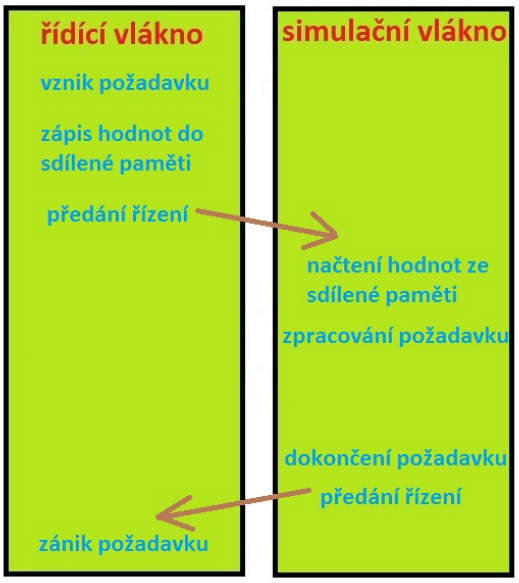
V implementaci existují dva synchronizační mechanismy. Jedná se o synchronizaci inicializace obou vláken a potom o předávání žádosti o změnu simulace. Ve skutečnosti běží vláken v programu více, protože implementace JTAG adaptéru je rovněž vícevláknová. S těmito vlákny tato práce ovšem nepracuje.

Inicializace vláken je synchronizována pomocí obyčejného mutexu. Tento mutex je zamknutý ještě před vytvořením řídícího vlákna. Řídící vlákno poté čeká na odemknutí tohoto mutexu než začne posílat data do pluginu. Mutex je uvolněn simulačním vláknem po ukončení inicializace. Pokud je inicializace simulace hotová rychleji než inicializace v řídícím vlákně, je možné, že nějakou dobu poběží simulace bez toho, aby přijímala data od, zatím ještě neinicializovaného, řídícího vlákna. Tedy při úspěšné inicializaci obou vláken zůstává tento mutex zamknut až do konce programu. Synchronizace inicializace je zobrazena na obrázku 4.3.



Obrázek 4.3: Schéma průběhu komunikace mezi vlákny při inicializaci

Žádost o změnu dat v simulaci je synchronizována pomocí podmínkové proměnné a SRW zámku. Každý zahájený požadavek nastaví příznak provádění operace, který může simulační vlákno přečíst, a čeká, dokud nedostane zprávu o vyřízení požadavku pomocí podmínkové proměnné. Simulační vlákno periodicky kontroluje nastavení příznaku provádění a pokud je nastaven, začne provádět operace nad simulací. Po ukončení všech operací je příznak provádění nastaven na hodnotu, která uvolňuje podmínkovou proměnnou a je poslána notifikace pro podmínkovou proměnnou, na kterou čeká řídící vlákno. Tím se zakáží další změny v simulaci a zároveň řídící vlákno pokračuje ve své činnosti. Grafické znázornění této synchronizace je uvedeno na obrázku 4.4.



Obrázek 4.4: Schéma průběhu komunikace mezi vlákny při požadavku na změnu simulace

Existuje několik způsobů, jak může být program ukončen:

1. Řídící vlákno může být ukončeno samo při vnitřní chybě. V takovém případě se vypíše chybová hláška. Obě vlákna jsou následně spojena jakmile se provede další tik signálu CLK uvnitř simulace a následně je program se simulací ukončen.

2. Pokud nastane chyba v simulačním vlákně, je řídícímu vláknu předán příkaz k ukončení. Samotné spojení vláken se poté děje stejným způsobem jako v situaci 1 a rovněž se vypíše chybová hláška.

3. V případě, že uživatel pošle příkaz k ukončení řídícího vlákna, je ukončeno tentokrát bez vypsání chybové hlášky. Spojení vláken probíhá stejným způsobem jako v situaci 1.

4. Pokud existuje v okamžiku chyby pouze jedno vlákno, je program ukončen korektně bez jakékoliv nutnosti synchronizace vláken. Takový případ nastane před zavoláním funkce `Start` (popřípadě, pokud tato funkce vrátí `false`) nebo po spojení obou vláken ve funkci `m_WaitForDebuggerFunc`.

5. Pokud nastanou chyby zároveň v simulačním i řídícím vlákně, měly by být vždy korektně uvolněny všechny zdroje, ale je možné, že se simulátor neukončí korektně nebo že budou vypsány nesprávné chybové hlášky.

6. Jediný neošetřený chybový případ je při selhání synchronizačních mechanismů mezi vlákny. V některých případech jsou alokované zdroje alespoň částečně uvolněny. V takovéto situaci se program nemusí korektně ukončit ani korektně uvolnit alokované zdroje. Uživatel je ale o tomto případu informován chybovou hláškou vysvětlující situaci.

4.5 Implementace RTL pluginu

Implementace RTL pluginu se zkládá z pěti souborů: *rtl_plugin.h*, *rtl_plugin.cpp*, *rtl_main.cpp*, *rtl_config.h*, *rtl_config.cpp*. Veškeré implementované funkce, objekty i proměnné jsou uzavřeny v namespace `codasip::simulator::jtag`.

4.5.1 rtl_main.cpp

Tento soubor obsahuje definice funkcí, které jsou požadované pluginovým systémem JTAG adaptéru. Tyto funkce pouze částečně kontrolují parametry funkcí a zajišťují korektní přeposílání chyb do JTAG adaptéru. Samotná funkcionalita je implementována v objektu `RtlPlugin` v dalších souborech. Funkce pro vytváření a zrušení pluginu pouze vytvářejí nebo ruší tento objekt. Ukazatel na používaný objekt je uveden jako první parametr většiny funkcí. Podle tohoto parametru se předává řízení programu přímo do metod objektu. Dále obsahuje tento soubor strukturu, která zapouzdřuje tyto funkce a další informace o RTL pluginu, což je také požadováno JTAG adaptérem. Poslední věc je vytvoření instance třídy `StaticPluginRegister`. Definice této třídy je součástí JTAG adaptéru a je deklarována v hlavičkovém souboru *plugin_manager.h*. Parametrem konstruktoru této třídy je zmiňovaná struktura s informacemi o pluginu. Vytvořením instance této třídy se do JTAG adaptéru na globální úrovni programu přidá odkaz na nový plugin a je možné tento plugin vybrat při jeho zapnutí pomocí argumentu programu nebo, jako v našem případě, pomocí předvytvořených argumentů v konfiguračním souboru. Tento způsob je nutné zvolit, protože základní JTAG adaptér je samostatně spustitelný program, a i když je možné použít ho jako knihovnu, je nutné specifikovat jeho funkcionalitu pomocí parametrů stejně jako v samostatném programu. Tyto parametry jsou načítány ze souboru, kam je uživatel musí zadat. Cesta k tomuto souboru je poté předána do programu pomocí proměnné prostředí s názvem *CODASIP_JTAG_ARGS_FILE*.

4.5.2 rtl_plugin.h

V tomto souboru se nachází deklarace třídy `RtlPlugin`. Tento soubor obsahuje korespondující funkci ke všem funkcím v souboru *rtl_main.cpp*. Pouze parametr určující objekt pro vykonání funkce není přítomen, protože již byl použit pro zavolání těchto funkcí. Dále tento objekt obsahuje několik statických funkcí, například pro ukončení programu nebo načtení a zpracování argumentů pro JTAG adaptér. Tyto funkce musí být statické, protože se volají ještě před inicializací JTAG adaptéru. A instanci RTL pluginu vytváří právě až JTAG adaptér při své inicializaci. Další důležité funkce slouží pro zpracování přijatých dat. Jsou zde funkce například pro provedení jednoho tiku nebo pro inicializaci JTAG rozhraní. V neposlední řadě obsahuje tento objekt i pomocné proměnné a především veškeré nutné synchronizační mechanismy mezi vlákny (viz kapitola 4.4).

4.5.3 rtl_plugin.cpp

Tento soubor obsahuje implementaci třídy `RtlPlugin`. Jsou zde definovány počáteční hodnoty statických proměnných a definice všech metod třídy. Navíc jsou zde definovány funkce, které jsou určené pro použití v podpluginech (viz kapitola 4.3). Tyto funkce se do pluginů předávají pomocí ukazatelů přes strukturu. Proto nemohou být součástí třídy `RtlPlugin`. Ke každému ukazateli uvnitř předávané struktury je definována funkce, která implementuje funkcionalitu danou popisem této funkce při deklaraci ukazatele v souboru *jtag_*

plugin.h. Poslední funkce, kterou tento soubor obsahuje, je funkce `Start`, která byla již několikrát v této práci zmiňována. Tato funkce je zodpovědná za předání naplněné struktury do pluginu, vytvoření daného pluginu, načtení argumentů pro JTAG adaptér ze souboru a vytvoření řídícího vlákna programu. Toto je jediná funkce RTL pluginu, která musí být exportována z výsledné knihovny. Musí tedy být uzavřena v bloku `extern"C"`, kvůli správnému linkování knihovny k jednotlivým pluginům. Dále musí být v knihovně viditelná pro jednotlivé pluginy. Toho je dosaženo použitím makra `__declspec(dllexport)` při překladu pomocí Visual Studia nebo makra `__attribute__((visibility("default")))` pro překlad pomocí MinGW překladače.

4.5.4 rtl_config.cpp a rtl_config.h

Tyto soubory obsahují deklaraci a definici třídy, která je zodpovědná za načítání konfigurace RTL pluginu. Tato konfigurace je uložená ve formátu XML a její příklad je uložen na přiloženém paměťovém médiu a v příloze A.2. Funkce třídy spočívá v tom, že otevře soubor s konfigurací a začne ho parsovat pomocí programu *tinyxml2* [7]. Ošetřuje většinu chyb, které mohou vzniknout při čtení souboru a zpracování jeho obsahu. Dále poté načítá jednotlivé atributy, které jsou pro RTL plugin povoleny. Aktuálně je povolen pouze jeden atribut, který určuje instrukci zpřístupňující Nexus rozhraní modelu poté, co je nahrána do JTAG rozhraní. Tento atribut se nazývá *USERCODE*. Po korektním načtení konfigurace umožňuje tato třída získání hodnoty jednotlivých konfiguračních položek pomocí svých metod.

4.6 Implementace Questa pluginu

Podobně jako RTL plugin, má i Questa plugin rozdělenou implementaci do několika souborů. Jsou to *questa_plugin.h*, *questa_plugin.cpp* a *questa_main.cpp* Veškerý kód je uzavřen v namespace `codasip::simulator::jtag::rtl`.

4.6.1 questa_main.cpp

Tento soubor, podobně jako `rtl_main.cpp`, obsahuje všechny funkce, které potřebuje RTL plugin provolávat z Questa pluginu (viz kapitola 4.3). Tyto funkce pouze částečně kontrolují parametry a provolávají funkce třídy `QuestaPlugin` z dalších souborů. Zajišťují správné přeposílání chyb do RTL pluginu a tedy i do JTAG adaptéru. Také je zde struktura na globální úrovni s informacemi o pluginu, která se posílá do RTL pluginu. Narozdíl od RTL pluginu není potřeba Questa plugin registrovat na globální úrovni. Předpokládá se, že se program vždy spustí pouze pomocí jediného konkrétního pluginu, který má právě jednu struktru s informacemi a není tedy možné zaregistrovat pluginů více.

V tomto souboru jsou také umístěné funkce, které přímo ovlivňují simulaci. Implementace těchto funkcí uvnitř třídy by mohla způsobit některé přístupové problémy. Speciálně funkce, která se provolává bezprostředně po nahrání vytvořené knihovny do simulátoru, musí být na globální úrovni. Tato funkce je zodpovědná za přípravu simulace a za inicializaci RTL pluginu a JTAG adaptéru. Protože v okamžiku vykonávání této funkce ještě není nahrán hardwarový model do simulace, je třeba rozdělit inicializaci do dvou částí. Druhá část inicializace se provádí až po nahrání modelu a přípravě simulace uvnitř simulátoru. Při této události se vyvolá callback, který byl zaregistrován v první části simulace.

Nyní už je možné najít signál CLK uvnitř modelu a nastavit funkci, která se bude volat při jeho změně. Tato funkce je jednou z nejdůležitějších částí programu.

Funkce, která se pravidelně vykonává při změně CLK signálu v simulaci má několik důležitých součástí. Předně kontroluje, zda ještě existuje řídící vlákno programu a pokud ne, simulaci příslušně ukončí. Poté kontroluje, zda byl z řídícího vlákna poslán požadavek na změnu simulace. Pokud se toto nestalo, funkce skončí bez dalších operací. Pokud je zaznamenán nový požadavek od RTL pluginu, je po několik dalších provolání této funkce zpracováván. Aktuální stav se uchovává pomocí statických proměnných. V první fázi se nastaví v simulaci nové hodnoty signálů TMS a TDI a také se zajistí nastavení portu TRST na hodnotu logická 1. Pokud by tento signál změnil svou hodnotu z logické 1 na logickou 0, došlo by k resetu TAP kontroléru uvnitř modelu. Ve druhé fázi se změní hodnota signálu TCK na hodnotu logická 1. Tímto okamžikem začne TAP kontrolér zpracovávat nové hodnoty signálů, které byly nastaveny dříve. Třetí fáze je nastavení signálu TCK zpět na logickou 0, a tím ukončení jednoho tiku TAP kontroléru. V této chvíli je také k dispozici nová hodnota signálu TDO, kterou je třeba ze simulace přečíst a vrátit jí zpět do RTL pluginu jako výsledek požadavku. Tyto tři fáze mezi sebou vyžadují určité zpoždění, aby JTAG rozhraní v modelu stihlo zpracovat všechny potřebné informace a nedošlo ke ztrátě dat. Tato zpoždění jsou definována jako konstanty na začátku tohoto souboru.

Pro čtení hodnot ze simulace jsem vytvořil novou funkci, která na základě názvu signálu, který je předáván parametrem, je schopná zjistit aktuální hodnotu signálu v simulaci. K tomu je třeba nejprve signál najít, poté získat údaje o jeho současném stavu a nakonec z těchto údajů vybrat hodnotu signálu. Toho je dosaženo pomocí funkcí z rozhraní VPI (viz kapitola 2.2.1). Hodnota signálu je získána z funkce jako její návratová hodnota. Podobná funkce existuje i pro zápis nové hodnoty signálu. Tato funkce má navíc jeden parametr, který určuje novou hodnotu. Návratovým typem této funkce je `void`. Princip je podobný jako u předchozí funkce, pouze se nečte hodnota signálu, ale je přepsána novou hodnotou a uložena zpět do signálu.

4.6.2 questa_plugin.h a questa_plugin.cpp

V těchto souborech je umístěna deklarace a definice třídy `QuestaPlugin`. Tato třída je podobná stejnému typu třídy v RTL pluginu. Stejně jako tam jsou i zde definovány funkce, které vykonávají příkazy poslané z nadřazeného objektu (v tomto případě z RTL pluginu). Je zde také proměnná se statickou referencí na aktuální objekt této třídy, což se podobá návrhovému vzoru singleton. Důležitou součástí instance této třídy je, že uchovává proměnné, které je třeba předat z řídícího vlákna programu do simulačního vlákna. Jedná se o hodnoty vstupních i výstupních signálů TAP kontroléru. Funkce, která zpracovává požadavek z RTL pluginu vždy nastaví tyto proměnné na nové hodnoty signálů, signalizuje simulaci, která provede požadavek, jak je naznačeno v předchozí kapitole, a poté vrátí proměnnou, kterou simulace nastavila na hodnotu výstupního signulu TDO.

Tato třída také obsahuje implementaci funkce, která je volána při ukončování programu a je zodpovědná za jeho korektní ukončení. Tuto funkci lze zavolat z obou vláken programu. Rozdíl je pouze v parametru, podle kterého funkce rozhoduje, jaké části programu má ukončit. V obou případech je zpracována chybová hláška, která je také předána parametrem. Ta je vytisknuta pomocí funkce z VPI, která obstarává tisk na standardní simulační výstup. Pokud byla funkce zavolána z řídícího vlákna, je instance pluginu zrušena a poté je funkce ukončena. To způsobí, že řídící vlákno je následně ukončeno a tato funkce znovu zavolána tentokrát ze simulačního vlákna. Pokud je tato funkce zavolána ze simulačního vlákna,

ukončí se aktuální instance pluginu, pokud již nebyla zrušena dříve. Dále se čeká na ukončení řídícího vlákna programu. Toto vlákno by mělo být již ukončené nebo by mělo být v procesu ukončování. Nakonec se pošle příkaz simulaci k zastavení a tím se program ukončí.

Kapitola 5

Spouštění a testování

Překlad programu je popsán v souboru *README.txt* na přiloženém paměťovém médiu. Výsledkem překladu jsou dvě knihovny s názvy *rtljtagadapter.dll* a *questaplugin.dll*. Při spouštění je nutné předat do simulátoru cestu k souboru *questapligin.dll* a cestu k souboru *rtljtagadapter.dll* mít uloženou v proměnné prostředí *PATH*, protože je k programu připojena až za běhu. Stejným způsobem je potřeba ošetřit všechny knihovny, které JTAG adaptér potřebuje při běhu. Nejjednoduším způsobem je použít program Codasip Studio, jehož je JTAG adaptér součástí, a který obsahuje všechny potřebné knihovny pro spuštění.

Obsah přiloženého paměťového média je uveden v příloze B. Příklad spuštění programu je uveden v příloze C.

5.1 Prerekvizity

Ke správnému fungovaní tohoto programu je třeba několik komponent.

- Questa Advanced Simulátor je program, na kterém tato knihovna závisí. Ke správnému překladu je třeba použít hlavičkové soubory *vpi_user.h* a *vpi_compatibility.h*. Oba tyto soubory poskytuje Questa Advanced Simulátor a pro účely této práce jsou přiloženy ke zdrojovým kódům. Rovněž je nutné za překladu linkovat knihovnu *mtipli*. Samostatné spuštění programu není možné. Jediný způsob jak knihovnu použít, je předat cestu k ní do simulátoru, který si danou knihovnu sám spustí (viz kapitola 2.2.1).

- RTL model v jazyce Verilog je nutnou součástí spuštění programu. Tento model by měl obsahovat JTAG a Nexus rozhraní, na které se JTAG adaptér umí napojit. Model také musí obsahovat soubor *nexus_config.xml*, který obsahuje základní informace o RTL modelu a který je nutný pro správnou komunikaci mezi JTAG adaptérem a modelem.

- Konfigurační soubory pro RTL plugin a JTAG adaptér musí být také přítomny. Konfigurace RTL pluginu je většinou uložena v souboru *rtl_config.xml* (viz kapitola 4.5.4). Konfigurace pro JTAG adaptér je soubor obsahující argumenty pro samostatně spustitelnou verzi JTAG adaptéru. Tento soubor je načítán při inicializaci RTL pluginu a jeho obsah je přeposlán ve formě parametrů do JTAG adaptéru.

- Testbench je nutný pro běh modelu uvnitř simulace. Měl by být napsaný v jazyce Verilog a neměl by zastavovat ani jinak v průběhu ovlivňovat simulaci po její inicializaci. Součástí také často bývá spouštěcí script (například ve formátu TCL), který

dokáže automaticky přeložit zdrojové soubory modelu a testbench, nakonfigurovat simulaci a zahájit ji.

- Dále potřebujeme přeloženou aplikaci pro daný RTL model procesoru, která bude nahrána do modelu na začátku simulace a která bude běžet na procesoru po dobu simulace.

- Poslední nutnou součástí je program, který dokáže ovládat JTAG adaptér a tím vytvářet požadavky na změnu simulace. Toho je schopný například debugger v programu Eclipse nebo v programu Codasip Studio. JTAG adaptér naslouchá na portu, který je mu dán jako parametr při inicializaci. Podporovaný debugger se poté připojí na tento port a začne posílat příkazy, kterým JTAG adaptér rozumí a tím ovlivňuje běh simulace.

5.2 Testování

Tento program byl určený pro překlad pomocí překladače MinGW. Zdrojové soubory by mělo být možné přeložit i pomocí Visual Studia, ale tento způsob není otestován. Výsledná knihovna byla otestována za pomoci programu Questa Advanced Simulator ve verzích 10.5b a 10.6 pro 64 bitové systémy. Pro správnou funkcionalitu je nutné použít verzi simulátoru, která obsahuje vlastní překladač gcc, z důvodu jistých nekompatibilit při překladu pomocí MinGW. Veškeré testování probíhalo na platformách Windows 8.1 a Windows 10. Testovací debugger, určený ke komunikaci se simulací, je součástí produktu Codasip Studio 6.2.4. Několik procesorových jader bylo využito při testování. Všechna tato jádra jsou pod licencí společnosti Codasip a nelze je tedy v této práci uvádět jako příklady.

Testování odhalilo několik chyb, které byly následně opraveny. Další testování probíhalo částečně ručně a částečně automatizovaně a bylo dokončeno úspěšně bez nalezení dalších závažných chyb či nedostatků.

5.3 Použití Codasip Studia jako debuggeru

Nejjednodušším způsobem, jak se napojit na běžící JTAG adaptér a simulaci je použít debugger vestavěný v produktu Codasip Studio. Nejprve je třeba otevřít okno *run -> debug configurations ...* a zvolit položku *codasip attach to process*. Tím vytvoříme novou konfiguraci pro debugger. Této konfiguraci je třeba nastavit položku *port* na číslo portu zadané v argumentech JTAG adaptéru a položku *host* na hodnotu *localhost*. Po spuštění debuggeru tlačítkem *Debug* a přepnutí do debuggovací perspektivy začne probíhat další komunikace mezi JTAG adaptérem a simulací. V této fázi se čtou ze simulace data potřebná pro zobrazení v debuggeru. Aplikace je poté zastavena na své první instrukci. Po dokončení této komunikace je debugger připraven přijímat příkazy od uživatele a aplikovat je pomocí JTAG adaptéru, RTL pluginu a Questa pluginu přímo na běžící simulaci.

Tento debugger podporuje širokou škálu funkcí. Níže jsou uvedeny některé příklady.

- Zjištění aktuálního stavu a změna aktuálního stavu registrů v modelu.

- Zjištění aktuálního stavu a změna aktuálního stavu paměti v modelu.

- Zjištění aktuálního stavu a změna aktuálního stavu proměnných v aktuální funkci (pouze pro aplikaci v jazyku C).

- Provedení jedné instrukce assebleru.
- Krokování pomocí příkazů *step over*, *step into*, *step return* a *step out*. (pouze pro aplikaci v jazyku C).
- Pokračování simulace.
- Pozastavení simulace.
- Zastavení simulace.
- Zapnutí a vypnutí breakpointu pro konkrétní řádek programu.

Aplikace, která běží v simulaci, může být přeložená buď pomocí překladače pro assembler nebo pro jazyk C závislým na konkrétním procesorovém modelu. Pokud je k dispozici knihovna obsahující disassembler procesoru, jsou jednotlivé instrukce programu viditelné uvnitř debuggeru. Pokud se jedná o aplikaci napsanou v jazyku C, je možné nahrát do debuggeru zdrojový kód aplikace a krokovat aplikaci přímo v jazyce C. Je zajištěno, že po zastavení debuggeru je ukončena i simulace a při ukončení simulace je zastaven debugger.

Kapitola 6

Závěr

Má implementace zadaného problému je otestována a plně funkční. Jedná se o rozšíření již existujícího produktu společnosti Codasip, JTAG adaptéru. I když je tato implementace závislá na již vytvořeném produktu, návrh implementace na tomto produktu závislý není. JTAG adaptér od společnosti Codasip jsem použil především pro možnost otestování mého návrhu. Myšlenka a návrh, který jsem vytvořil, je použitelná obecně pro programy podobného typu. Jediná nutná úprava této implementace při použití s jinými programy je změna vstupního rozhraní pro RTL plugin.

Můj přístup má několik výhod. Především je moje implementace snadno rozšiřitelná. Pluginový systém, který jsem vytvořil pro RTL plugin, podporuje implementaci většího množství pluginů podobného typu jako Questa plugin. To znamená, že by se tento projekt dal rozšířit o jiná komunikační rozhraní uvedená v kapitole 2.2. A jelikož není moje implementace závislá na konkrétním RTL simulátoru, mělo by být možné drobnými úpravami použít vytvořený plugin i pro jiné simulátory (např. Riviera-PRO nebo VCS Simulator). Verzovací systém pluginů zajišťuje možnost upravovat rozhraní pluginů podle potřeby, aniž by docházelo k neočekávaným nebo nespecifikovaným stavům programu. Toho lze využít například pokud by bylo třeba doimplementovat novou funkcionalitu programu. Samotný RTL plugin je také připraven pro možnost přidávat nové funkcionality. Toho se dá dosáhnout například přidáním dalších argumentů JTAG adaptéru specifických pro RTL plugin nebo pomocí konfiguračního souboru pro RTL plugin, který může obsahovat více informací potřebných k práci s konkrétním modelem.

Jednou z mála nevýhod této implementace je nepříliš velká rychlost programu. Kvůli rozdělování požadavků na jednotlivé bity se značně zpomaluje samotné ovlivňování simulace. Rychlost programu by bylo možné vylepšit posíláním větších kusů informací najednou do Questa pluginu, který by je dokázal efektivněji zpracovávat. Takovýto postup by ale znamenal komplexnější algoritmus pro posílání dat.

Tato práce má velké využití v oblasti testování návrhů nových procesorových jader. Umožňuje kombinaci debuggování jak hardwarového popisu tak softwarové aplikace zároveň. Oproti ostatním pluginům JTAG adaptéru, které jsou určeny pro použití s reálným hardwarem, pracuje tento pouze se simulací hardwaru. To znamená, že se nemusí hardware vyrábět a některé vady designu nebo nedostatky je možné objevit dříve a tudíž ušetřit peníze za výrobu testovacího hardwaru. Použití této práce je výhodné také v tom, že umožňuje uživateli otestovat hardware v simulaci přímo s konkrétní aplikací, která má uvnitř hardwaru běžet.

Literatura

[1] *Freescale Nexus 5001 Software Debug Interfaces* .
URL http://www.ip-extreme.com/IP/nexus_5001.shtml

[2] *ModelSim® SE User's Manual & Command Reference - vsim* .
URL http://www.pldworld.com/_hdl/2/_ref/se_html/manual_html/c_vcmds191.html#11471491

[3] *Questa® Advanced Simulator* .
URL https://www.mentor.com/products/fv/questa/

[4] *Riviera-PRO™* .
URL
https://www.aldec.com/en/products/functional_verification/riviera-pro

[5] *SystemVerilog DPI Tutorial* .
URL https://www.doulos.com/knowhow/sysverilog/tutorial/dpi/

[6] *Technical Guide to JTAG* .
URL https://www.xjtag.com/about-jtag/jtag-a-technical-overview/

[7] *TinyXML-2* .
URL https://github.com/leethomason/tinyxml2

[8] *VHPI Applications* .
URL
https://www.aldec.com/en/support/resources/documentation/articles/1457

[9] *What is JTAG and how can I make use of it?* .
URL https://www.xjtag.com/about-jtag/what-is-jtag/

[10] *IEEE Standard Verilog ® Hardware Description Language*. Září 2001.
URL https://inst.eecs.berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf

[11] *ModelSim® Foreign Language Interface Version 5.6d* . Srpen 2002.
URL http://homepages.cae.wisc.edu/~ece554/new_website/ToolDoc/Modelsim_docs/docs/pdf/fli.pdf

[12] *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language* . Únor 2013.

[13] *Codasip Studio Technical Reference Manual - Version: 6.2.4* . Leden 2017.
URL https://support.codasip.com/downloads/docs/

[14] *Codasip Studio User Guide - Version: 6.2.4* . Leden 2017.
URL https://support.codasip.com/downloads/docs/

[15] Foster, H.: *Part 10: The 2016 Wilson Research Group Functional Verification Study - ASIC/IC Language and Library Adoption Trends* . Říjen 2016.
URL https://blogs.mentor.com/verificationhorizons/blog/2016/10/31/part-10-the-2016-wilson-research-group-functional-verification-study/

[16] Sutherland, S.: *The Verilog PLI Handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface*. The Springer International Series in Engineering and Computer Science, 2006, ISBN 9780306476655.

Přílohy

Příloha A

Ukázkové konfigurační soubory

A.1 rtl_config.xml

```
<RTL>
    <CHAIN USERCODE = "0011" />
</RTL>
```

A.2 jtag_args.cfg

```
--plugin rtl
-C C:\Path\To\Root\rtl\rtl_config.xml
-c C:\Path\To\Root\rtl\nexus_config.xml
-p 40000
-i mi
-a procesor_core_name C:\Path\To\Root\app\application.exe --load
```

Příloha B

Obsah přiloženého paměťového média

- Všechny soubory, které obsahují moji vlastní implementaci jsou uloženy v adresáři *src*.
- Soubory *tinyxml2.cpp* a *tinyxml.h*, které obsahují implementaci pro zpracování XML souborů[7], jsou uloženy v adresáři *tinyxml*
- Všechny hlavičkové soubory nutné pro JTAG adaptér a VPI rozhraní jsou uloženy v adresáři *include*. Soubor *jtag_plugin.h*, který je součástí tohoto adresáře, je z části napsaný mnou a z části společností Codasip. Ostatní soubory jsou převzaty z příslušných programů.
- Ukázkové konfigurační soubory pro JTAG adaptér a RTL plugin jsou uloženy v adresáři *examples*. Tyto soubory jsou také k dispozici v přílohách A.1 a A.2.
- Spustitelná knihovna *libquestaplugin.dll* je uložena v adresáři *bin*. Tato knihovna neobsahuje implementaci RTL pluginu. Ta je začleněna v knihovně JTAG adaptéru, kterou v této práci nemohu zveřejnit kvůli její licenci.
- Zdrojový kód technické zprávy v jazyce Latex je uložen v adresáři *doc*.
- Technická zpráva ve finální podobě je uložena v hlavním adresáři v souboru *TechnickaZprava.pdf*.
- Soubor *README.txt*, obsahující obsah paměťového média a informace o překladu, je uložen v hlavním adresáři.

Příloha C

Příklad spuštění

Předpokládejme následující adresářovou strukturu:

- Adresář *bin*, ve kterém jsou umístěny přeložené knihovny *rtljtagadapter.dll* a *questaplugin.dll* a všechny knihovny, které jsou potřeba ke správnému spuštění JTAG adaptéru.
- Adresář *rtl*, ve kterém je umístěn RTL model určeného procesorového modelu v jazyce Verilog. Rovněž soubory *nexus_config.xml* a *rtl_config.xml* jsou umístěny v tomto adresáři.
- Adresář *tb*, ve kterém je umístěn připravený testbench v jazyce verilog společně s TCL scriptem, který dokáže Questa Adavenced Simulator zpracovat.
- Adresář *app*, ve kterém je umístěna přeložená aplikace *application.exe*, která má běžet na RTL modelu.

Důležitým předpokladem je, že daný RTL model je kompatibilní s JTAG adaptérem. Tedy, že obsahuje správně nakonfigurované JTAG i Nexus rozhraní. Popisy tohoto modelu v jeho konfiguračních souborech musí také odpovídat modelu. Dalším důležitým předpokladem jsou vlastnosti TCL scriptu uvnitř *tb* adresáře. Tento script by měl obsahovat cestu ke zdrojovým kódům uvnitř adresáře *rtl*. Tyto kódy jsou společně se souborem obsahujícím testbench přeloženy simulátorem. Tento script se také stará o inicializaci a zahájení simulace. Velice důležitou součástí tohoto scriptu je příkaz *vsim*, který musí být vykonán pro zahájení simulace. Příkaz musí obsahovat parametr `-pli "../bin/questaplugin.dll"`, který připojí námi vytvořenou knihovnu k simulaci. Také by mělo být zajištěno, že všechny signály simulace mají právo být měněny zvenčí modelu, jinak by VPI příkazy na změnu nebo čtení signálu neproběhly úspěšně.

První věc, kterou musí uživatel udělat, je nastavit proměnnou prostředí *PATH* tak, aby obsahovala cestu do odresáře *bin* v naší adresářové struktuře. Dále musí uživatel vytvořit soubor (například v hlavním adresáři), který bude obsahovat argumenty pro JTAG adaptér. Ukázka tohoto souboru pro aktuální příklad je k dispozici v příloze A.1 a na paměťovém médiu. Absolutní cestu k tomuto souboru je potřeba uložit do proměnné prostředí s názvem *CODASIP_JTAG_ARGS_FILE*. Aby souhlasily relativní cesty k souborům, je třeba spustit program z adresáře, ve kterém se nachází TCL script. Tedy v našem případě je nutné přepnout se do adresáře *tb*. Předpokládá se použití příkazové řádky nebo podobného nástroje. Následuje spuštění programu Questa Advanced Simulator. Protože ho musíme spouštět z adresáře *tb*, je pravděpodobně nutné specifikovat celou cestu k souboru *vsim.exe* uvnitř

adresářové struktury progrmu Questa Advanced Simulator. Jediným parametrem programu *vsim.exe* by měl být parametr *-do* následovaný názvem TCL scriptu v aktuálním adresáři. Tím by se měl Questa Advanced Simulator spustit, zahájit simulaci a zahájit komunikaci s JTAG adaptérem. V první fázi JTAG adaptér kontroluje správnost svých konfigurací a nahraje do simulace aplikaci uloženou v adresáři *app*. Cesta k této aplikaci je předána pomocí konfiguračního souboru s argumenty pro JTAG adaptér. Po nahrání aplikace simulace pokračuje, ale nejsou jí předávána žádná data. JTAG adaptér je v této fázi plně inicializován a čeká na připojení debuggeru.