VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR LADICÍHO NÁSTROJE NA ČIPU

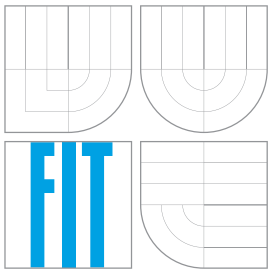
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK HRBÁČEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR LADICÍHO NÁSTROJE NA ČIPU

ON-CHIP DEBUGGER GENERATOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK HRBÁČEK

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací ladicího rozhraní procesoru a jeho napojení na hardware generovaný nástroji projektu Lissom. V práci jsou nejprve podrobně popsány standardy JTAG a Nexus 5001, kterým implementované rozhraní vyhovuje. Praktická část práce zahrnuje popis vyvinutého nástroje a jeho testování. Výsledkem práce je funkční ladicí nástroj otestovaný společně s procesorem Codea na výukové platformě FITKit.

Abstract

This bachelor's thesis deals with the design and implementation of an on-chip debugger and its connection to the hardware generated using software tools developed as a part of the Lissom project. The first part introduces the JTAG and Nexus 5001 standards, which the implemented interface complies with. The practical part includes detailed description of the developed tool and its testing. The result is a functional on-chip debugger that has been tested with the Codea processor on the FITKit educational platform.

Klíčová slova

ladění kódu, JTAG, Nexus 5001, Lissom, VHDL, vestavěný systém

Keywords

debugging, JTAG, Nexus 5001, Lissom, VHDL, embedded system

Citace

Radek Hrbáček: Generátor ladicího nástroje na čipu, bakalářská práce, Brno, FIT VUT v Brně, 2011

Generátor ladicího nástroje na čipu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc.

.....
Radek Hrbáček
14. května 2011

Poděkování

Na tomto místě chci poděkovat celému týmu, který se podílí na realizaci projektu Lissom, zejména vedoucímu mé práce prof. Ing. Tomáši Hruškovi, CSc., Ing. Karlu Masaříkovi, Ph.D. a Ing. Zdeňku Přikrylovi za pomoc při tvorbě bakalářské práce. Dále bych chtěl poděkovat své rodině za podporu během celého studia.

Tato práce vznikla za podpory grantu „FR-TI1/038 - Systém pro programování a realizaci vestavěných systémů“.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Při návrhu a realizaci vestavěných zařízení tvoří nezanedbatelnou část vývoje ladění kódu. Téměř každý na trhu dostupný procesor disponuje ladicím rozhraním, přičemž donedávna každý výrobce používal vlastní technologii rozhraní.

V dubnu roku 1998 bylo velkými výrobci, jako je např. Motorola (nyní Freescale Semiconductor), Texas Instruments nebo Ericson, založeno Nexus 5001™ Forum [1]. Výsledkem tohoto fóra se stal roku 2003 standard IEEE-ISTO 5001™-2003, který definuje škálovatelné ladicí rozhraní pro libovolný procesor.

Mým úkolem bylo implementovat podporu ladicího rozhraní podle standardu Nexus 5001 pro procesory popsané v jazyku ISAC (Instruction Set Architecture C) [4] a generované automatizovanými nástroji. Jazyk ISAC je vytvářen na Fakultě informačních technologií Vysokého učení technického v Brně v rámci projektu Lissom [5]. Cílem projektu je zjednodušit proces návrhu a verifikace (víceprocesorových) vestavěných zařízení vytvořením jazyka pro popis architektury procesoru a nástrojů, které dokáží automaticky generovat simulátor procesoru na různých úrovních detailů architektury, assembler, disassembler, linker, kompilátor a dekompilátor jazyka C a také hardwarovou reprezentaci syntetizovatelnou standardními nástroji. Důraz je kladen především na co největší shodu chování simulátoru a generovaného hardware, proto i návrh ladicího rozhraní musí co nejvíce odpovídat možnostem ladění kódu v simulátoru a naopak.

Obsahem této práce je nejprve detailní popis standardů JTAG (IEEE 1149.1) a Nexus 5001 (IEEE-ISTO 5001™-2003) a zbytek práce tvoří popis implementace rozhraní JTAG a ladicího nástroje, návrhu JTAG adaptéru a testovaní.

Kapitola 2

Standard JTAG (IEEE 1149.1)

Standard JTAG (Joint Test Action Group), definovaný normou IEEE 1149.1, vznikal od roku 1985 a byl poprvé vydán roku 1990. Jeho původním účelem bylo především testování plošných spojů, ale postupem času se uchytil i v jiných aplikacích, jako např. programování FLASH pamětí apod. Rozšíření JTAGu mimo jiné pomohla firma Intel, když roku 1989 uvedla na trh procesor Intel 80486, první procesor s rozhraním JTAG. Od roku 1990 vzniklo několik aktualizací, mezi nimi např. IEEE 1149.1b-1994 (doplňuje standard o jazyk pro popis testovaných obvodů - BSDL), IEEE 1149.1-2001 nebo nejnovější IEEE 1149.7-2009 přidávající mnoho nové funkčnosti při zachování zpětné kompatibility s IEEE 1149.1-2001 [7].

Přestože JTAG sám o sobě neposkytuje žádnou podporu pro ladění, díky svému masovému rozšíření je ve velké míře používán jako komunikační prostředek mezi ladicím rozhraním procesoru a vývojovým prostředím.

2.1 Architektura JTAG Test Access Portu

Princip rozhraní JTAG je založeno na tzv. *Boundary Scanu*. Datová cesta uvnitř nebo i vně integrovaných obvodů je přerušena a do cesty jsou vloženy buňky (*Boundary Scan Cell*) pracující v několika režimech:

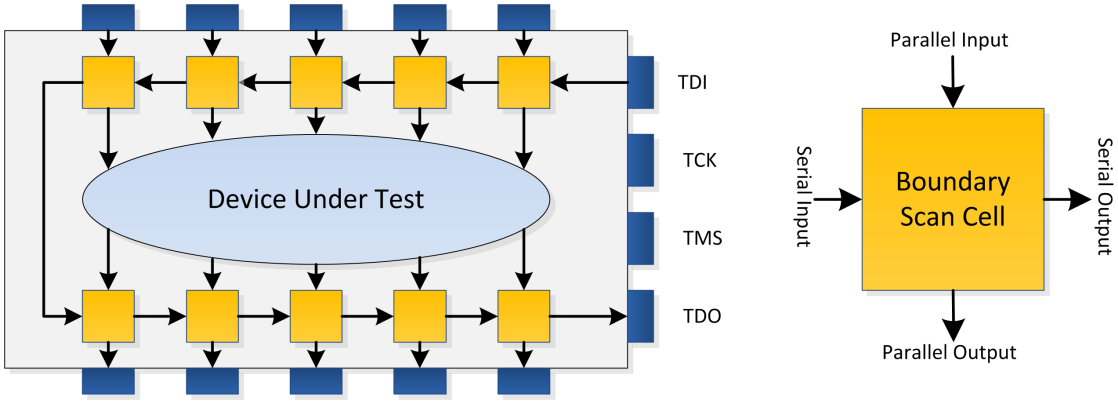
- propojení datové cesty
- zapsání požadované hodnoty na výstup
- přečtení aktuální hodnoty na vstupu

Tyto buňky jsou sériově propojeny a tvoří tak velký posuvný registr, do kterého jsou data nasouvána nebo jsou z něj naopak vysouvána. Společně s řadičem a několika dalšími obvody tvoří tzv. *Test Access Port* (TAP). Těch může být na desce plošných spojů resp. na čipu i několik. Veškerá komunikace probíhá synchronně pomocí pouze 4 povinných signálů a případně jednoho nepovinného:

- `TCK` *(Test Clock Input)* – hodinový signál, do rozhraní je zařazen především kvůli nezávislosti na hodinách jednotlivých testovaných komponent, kterých může být více s různými hodinovými signály
- `TMS` *(Test Mode Select Input)* – tento signál slouží k ovládání stavového automatu uvnitř řadiče TAP a tím k výběru testovacích operací.

- `TDI` *(Test Data Input)* – vstupní datový signál, hodnota je snímána na náběžné hraně `TCK`, data jsou nasouvány do datového nebo instrukčního registru.
- `TDO` *(Test Data Output)* – výstupní datový signál, hodnota je platná při sestupné hraně `TCK`.
- $\overline{\texttt{TRST}}$[1] *(Test Reset Input)* – nepovinný asynchronní reset aktivní v nule, resetu je možné dosáhnout i za pomocí signálů `TCK` a `TMS`.

Zapojení *Boundery Scan Cell* buněk je znázorněno na obrázku 2.1. Je zde vidět původní datová cesta vedoucí od pinů integrovaného obvodu k testované komponentě *(Device Under Test)* přerušená sériově zapojenými buňkami připojenými na rozhraní JTAGu, vpravo je pak detail jedné buňky.



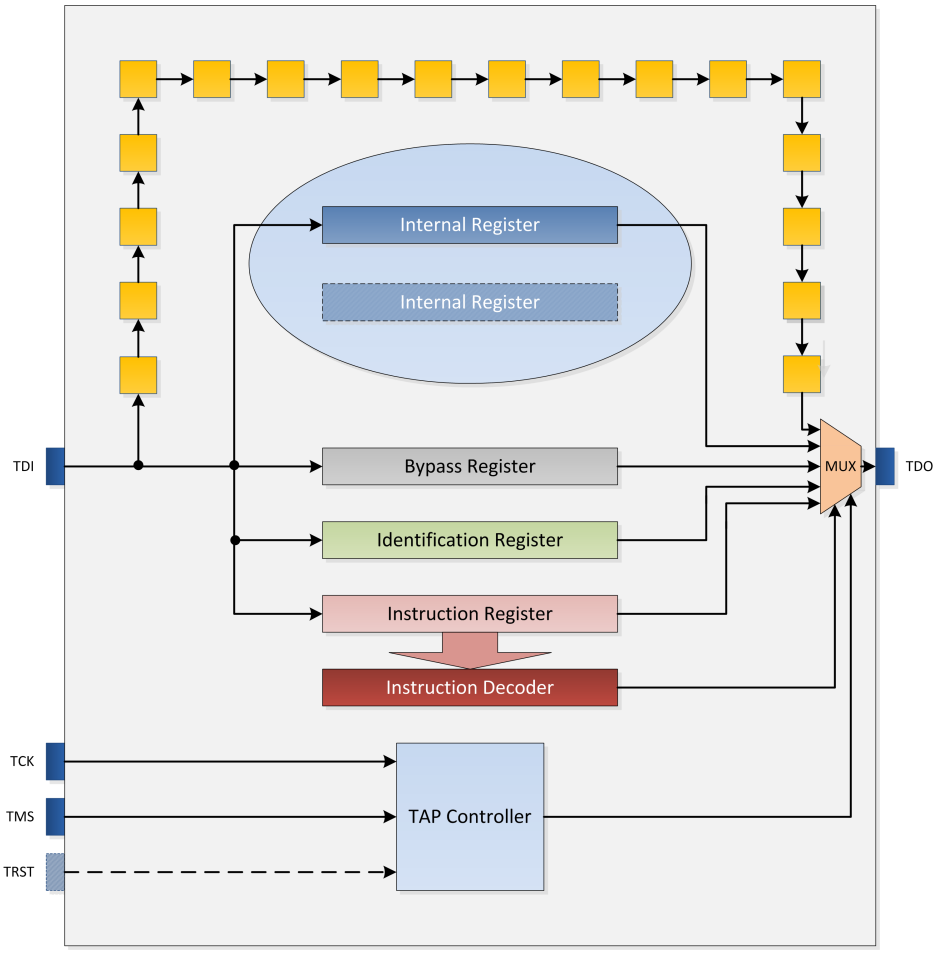
Obrázek 2.1: Princip *Boundary Scan* architektury

Celé rozhraní JTAG je složeno z celkem 7 částí:

- Řadič TAP *(TAP Controller)* – konečný stavový automat (viz 2.4), vstupem jsou signály `TCK` a `TMS` (příp. $\overline{\texttt{TRST}}$), výstupem řídicí signály vedoucí k registrům a multiplexorům.
- Instrukční registr *(Instruction Register)* – uchovává instrukci.
- Instrukční dekodér *(Instruction Decoder)* – dekóduje instrukci uloženou v instrukčním registru a nastavuje řídicí signály.
- Identifikační registr *(Identification Register)* – slouží k jednoznačné identifikaci zařízení.
- Bypass registr *(Bypass Register)* – umožňuje vybrat jen některé z více komponent zapojených do jednoho řetězce.
- Sada testovacích registrů *(Boundary Scan Registers)* – řetězce složené z elementárních buněk *Boundary Scan Cell* zapojené do různých datových cest.
- Výstupní multiplexory *(Output Multiplexers)* – vybírají výstup z testovacích registrů nebo instrukčního registru.

[1] Signály označené nadtržením jsou aktivní v nule.

Zapojení celého rozhraní včetně všech jmenovaných komponent je možné vidět na obrázku 2.2. V následujících podkapitolách jsou podrobněji popsány jednotlivé komponenty TAP.



Obrázek 2.2: Zapojení celého TAP

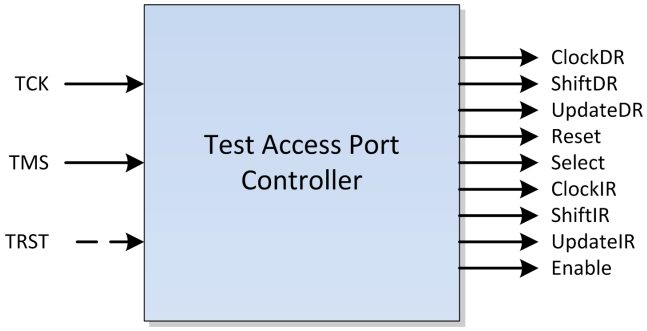
2.1.1 Řadič TAP

Úkolem řadiče TAP je nastavovat řídicí signály vedoucí k registrům a výstupním multiplexorům v závislosti na vstupních signálech `TCK` a `TMS`. Obrázek 2.3 zobrazuje vstupní a výstupní signály řadiče.

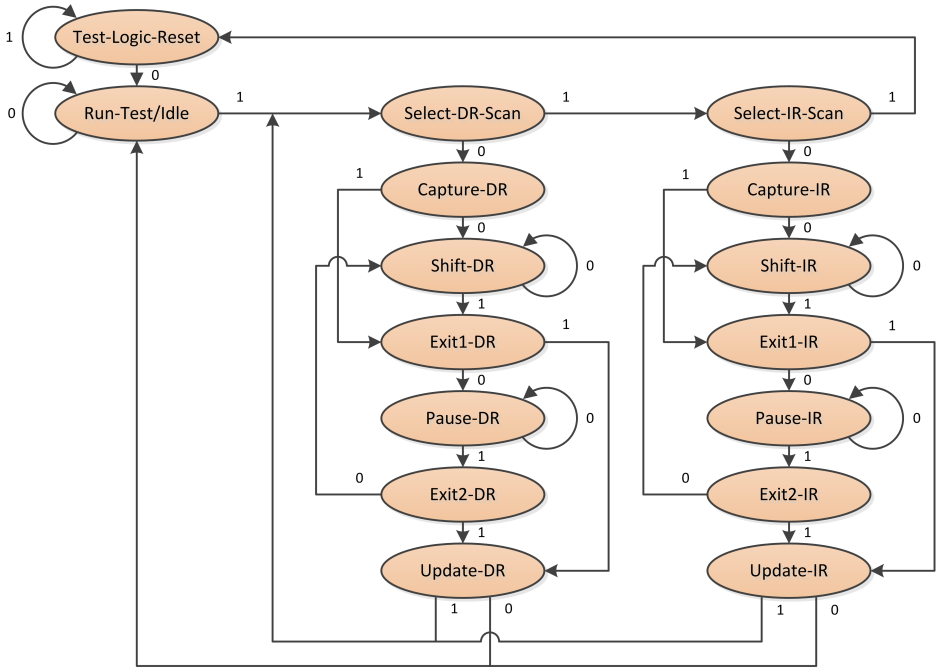
Řadič funguje jako synchronní konečný stavový automat, jeho stavy a přechody mezi nimi jsou ilustrovány na obrázku 2.4. Přechod mezi stavy nastává vždy na náběžnou hranu hodinového signálu `TCK`, následující stav je určen aktuálním stavem a signálem `TMS`.

Stav *Test-Logic-Reset* je jediný stav, ve kterém stavový automat zůstává při stabilní hodnotě 1 na signálu `TMS`, proto je možné z libovolného stavu řadič resetovat do tohoto stavu nejvýše 5 hodinovými cykly. Stejně tak lze ovšem použít nepovinný asynchronní signál $\overline{\texttt{TRST}}$, který při aktivní nule uvede automat právě do stavu *Test-Logic-Reset*.

Ve stavu *Run-Test/Idle* jsou aktivní testovací obvody, pokud je načtena odpovídající instrukce do instrukčního registru. Z tohoto stavu je také možné pokračovat na datový nebo



Obrázek 2.3: Vstupní a výstupní signály řadiče TAP



Obrázek 2.4: Stavy a přechody konečného stavového automatu uvnitř řadiče TAP

instrukční scan.

Select-DR-Scan a *Select-IR-Scan* jsou dočasné stavy řadiče, všechny datové registry zachovávají svou hodnotu. Z těchto stavů lze zahájit datový resp. instrukční scan nebo se vrátit do stavu *Test-Logic-Reset*.

Pokud se řadič nachází ve stavu *Capture-DR*, do posuvného datového registru jsou paralelně načtena data. V tomto stavu je na signál `ClockDR` přiveden hodinový signál `TCK`. Stav *Capture-IR* má podobný význam, ale s tím rozdílem, že se do posuvného registru nenačítají data, ale vzorek logických hodnot, typicky nějaké stavové signály. Na nultý bit (LSB) však musí být vložena hodnota 1 a na první bit nula. Díky tomu lze odhalit chybu v testovacím řetězci.

Ve stavu *ShiftDR* je aktivní signál `ShiftDR` a datový registr vybraný aktuální instrukcí je s hodinovým signálem `ClockDR` posouván. Analogicky je tomu i v případě stavu *ShiftIR* pro instrukční registr, v tomto případě je aktivní signál `ShiftIR`.

Stavy *Exit1-DR* a *Exit2-IR* jsou dočasné stavy řadiče, které umožňují buď ukončit scanovací sekvenci přechodem do stavu *Update-DR* resp. *Update-IR*, nebo ji pozastavit ve stavu *Pause-DR* resp. *Pause-IR*, ze kterého je následně možné se vrátit přes stav *Exit2-DR* resp. *Exit2-IR* do stavu *Shift-DR* resp. *Shift-IR*.

Zapsání sériově nasunutých dat do datových registrů je provedeno ve stavu *Update-DR* aktivováním signálu `UpdateDR`. Stav *Update-IR* zajišťuje zapsání nasunuté instrukce do instrukčního registru aktivováním signálu `UpdateIR`.

Signál `Enable` je aktivní, pokud se řadič nachází ve stavech *Shift-DR* nebo *Shift-IR*. Tento signál je přiveden na povolovací vstup výstupního třístavového budiče. Resetovací výstupní signál `Reset` je aktivován v případě, že se řadič nachází ve stavu *Test-Logic-Reset* nebo je aktivní vstupní signál $\overline{\texttt{TRST}}$. Poslední výstupní signál `Select` ovládá výstupní multiplexor, přepíná výstup instrukčního registru a multiplexovaného signálu z datových registrů.

2.1.2 Instrukční registr

Tento povinný registr umožňuje sériově nasunout instrukci signálem `TDI`, paralelně ji zapsat a zapamatovat. Na výstup `TDO` může být vysouván řetězec libovolných stavových signálů zakončený nulou a jedničkou. Velikost tohoto registru není předepsána, každá instrukce ale musí splňovat několik podmínek [3]:

- Každá instrukce určuje datový registr, který je zapojen do cesty mezi vstup `TDI` a výstup `TDO`.
- Ostatní datové registry nejsou aktivní a neovlivňují činnost TAP.
- Nevyužité kódy instrukcí musí být ekvivalentní instrukci BYPASS.

Standard JTAG dále definuje 3 povinné instrukce, které musí být implementovány v každém TAP:

- BYPASS – do cesty mezi vstup a výstup je zapojen bypass registr. Kód této instrukce je $11\ldots1$, tedy samé jedničky. Pokud TAP neobsahuje identifikační registr, jedná se o výchozí instrukci zapsanou do registru při resetu.
- EXTEST – datová cesta je přerušena v *Boundary Scan Cell* buňkách, při *CaptureDR* jsou do datového posuvného registru načteny hodnoty z paralelního vstupu, při *UpdateDR* je obsah posuvného registru zapsán na výstup buňky. Kód instrukce je $00\ldots0$, tedy samé nuly.
- SAMPLE – oproti instrukci EXTEST neovlivňuje datovou cestu, pouze umožňuje sledovat stav signálů přivedených na vstupy buněk. Kód instrukce je dán výrobcem.

Pokud TAP obsahuje také identifikační registr, musí implementovat také instrukci IDCODE, při které je mezi `TDI` a `TDO` zapojen identifikační registr. Dalšími nepovinnými instrukcemi jsou například INTEST, RUNBIST, USERCODE, CLAMP, HIGZ aj.

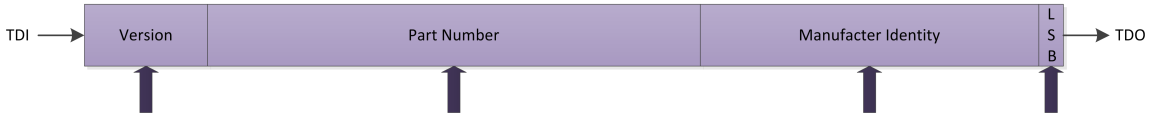
2.1.3 Instrukční dekodér

Vstupem instrukčního dekodéru je instrukce z instrukčního registru, výstupem pak především řídicí signál pro výstupní multiplexor přepínající výstupy jednotlivých datových registrů, dále pak další řídicí signály vedoucí k datovým registrům.

2.1.4 Identifikační registr

Nepovinný identifikační registr slouží k jednoznačné identifikaci testovaného zařízení. Jedná se o 32bitový registr rozdělený na 4 části:

- Bit 0 (LSB) – vždy 1 (kvůli automatickému rozpoznání TAP v řetězci).
- Bity 11-1 – *Manufacturer Identity* – označení výrobce zařízení, přiděluje organizace JEDEC.
- Bity 27-12 – *Part Number* – označení typu zařízení.
- Bity 31-28 – *Version* – specifikuje až 16 různých verzí jednoho zařízení.

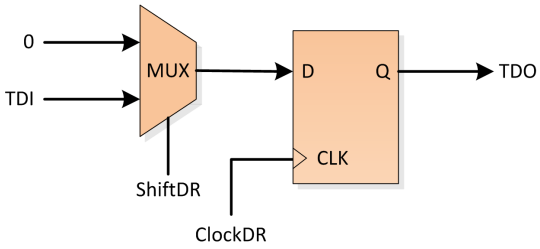


Obrázek 2.5: Struktura identifikačního registru

Na pozici nejméně významného bitu je vždy hodnota 1, aby bylo možné automaticky rozpoznat zařízení zapojené v řetězci za sebou. Pokud totiž TAP identifikační registr neobsahuje, výchozím datovým registrem se stává bypass registr, který je pouze jednobitový a má hodnotu 0. Identifikační řetězec bitů je do identifikačního registru načten pokaždé, když se řadič TAP nachází ve stavu *CaptureDR* a v instrukčním registru je načtena instrukce IDCODE.

2.1.5 Bypass registr

Bypass registr je povinný jednobitový datový registr, jehož paralelní vstup je nastaven na hodnotu 0 (viz obrázek 2.6). Slouží k zkrácení řetězce, pokud obsahuje více zařízení a je potřeba testovat pouze některá zařízení zapojená do řetězce. Tento registr je aktivní při instrukci BYPASS a také při každé hodnotě v instrukčním registru, které neodpovídá žádná známá instrukce.



Obrázek 2.6: Realizace bypass registru

2.1.6 Testovací registry

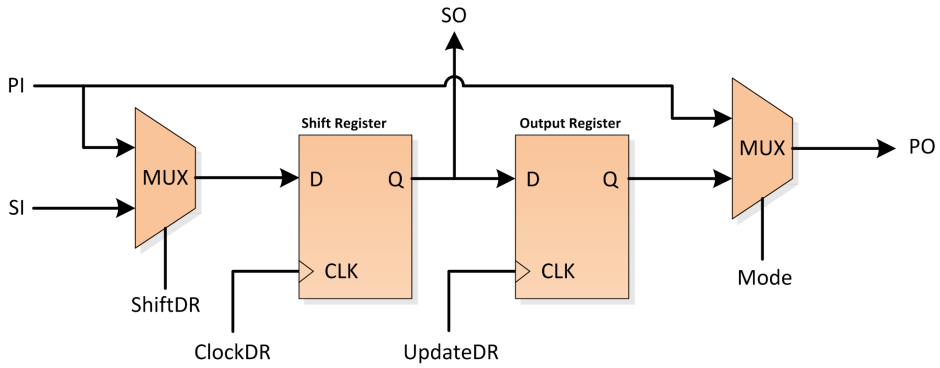
Každý TAP může obsahovat jeden nebo více testovacích registrů, tedy řetězců *Boundary Scan Cell* buněk, zapojených sériově mezi `TDI` a `TDO` a paralelně do různých datových cest. Výběr konkrétního registru je určen načtenou instrukcí.

Možné zapojení *Boundary Scan Cell* je možné vidět na obrázku 2.7, v tomto případě se jedná o vstupně-výstupní buňku. Sériový vstup `SI` je přiveden na vstupní multiplexor společně s paralelním vstupem `PI`, řídicím signálem je `ShiftDR` vedoucí z řadiče TAP. Výstup vstupního multiplexoru je přiveden na registr řízený hodinami `ClockDR`, jeho výstup je pak vyveden na sériový výstup `SO` a na výstupní registr. Spojením více buněk do řetězce pomocí sériových vstupů a výstupů vznikne posuvný registr. Signál z výstupního registru je dále veden na výstupní multiplexor, kde je přepínán s paralelním vstupem na paralelní výstup.

Pokud potřebujeme načíst data z paralelního vstupu a vysunout je ven ze zařízení, bude nejprve signál `ShiftDR` neaktivní (odpovídá stavu *Capture-DR*), tím dojde při nástupné hraně `ClockDR` k uložení hodnoty v posuvném registrum a poté při aktivním signálu `ShiftDR` je hodnota přenášena mezi buňkami sériovým kanálem.

Při náběžné hraně signálu `UpdateDR` (tedy při přechodu řadiče TAP do stavu *Update-DR*) je obsah posuvného registru nahrán do výstupního registru. Signálem `Mode` je pak řízen výstupní multiplexor, který určuje, zda bude buňka z hlediska datové cesty transparentní, nebo bude vnucovat požadovanou hodnotu na paralelním výstupu.

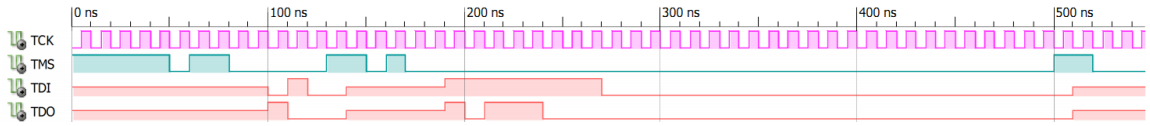
V závislosti na aplikaci je ale možné navrhnou i zcela odlišnou strukturu buňky, např. pouze s paralelním vstupem nebo výstupem. Nejnovější aktualizace standardu IEEE-1149.7-2009 přidává dokonce podporu pro diferenciální sběrnice a další nové možnosti.



Obrázek 2.7: Detail zapojení *Boundary Scan Cell* buňky

2.2 Časování signálů

Příklad komunikace přes rozhraní JTAG je uveden na obrázku 2.8. Nejprve je řadič TAP resetován pěti jedničkami na `TMS`, poté je sekvencí 01100 uveden do stavu *Shift-IR*, instrukce IDCODE (0010) je nasunuta ve čtyřech hodinových cyklech, poté je řadič uveden do stavu *Update-IR* a tím je nová instrukce zapsána do instrukčního registru. V další fázi je stav řadiče změněn na *Shift-DR* a na výstup `TDO` je vysouván identifikační řetězec (zde 00000000000000000000000000011101, tedy *Manufacturer* 00000001110, *PartNumber* 0000000000000000 a *Version* 0000).



Obrázek 2.8: Příklad komunikace přes JTAG – načtení instrukce IDCODE a přečtení identifikace zařízení

Standard JTAG nechává časování signálů na výrobci zařízení, nepředepisuje tedy žádné konkrétní hodnoty. Následující hodnoty ale musí být součástí dokumentace k zařízení [3]:

- maximální frekvence hodinového signálu `TCK`
- časové parametry testovací logiky
- logické úrovně pro vstupní (`TCK`, `TMS`, `TDI`) a výstupní (`TDO`) piny
- budicí schopnosti výstupu `TDO`

2.3 Aplikace

Kromě zařízení pracujících na nejnižší úrovni se můžeme setkat s JTAG rozhraním téměř ve všech vestavěných systémech. Slouží především k testování obvodů, ladění kódu a programování FLASH pamětí:

- architektura ARM podporuje JTAG, v některých případech mírně modifikovaný na dvousignálovou variantu SWD.
- FPGA a CPLD všech výrobců většinou používají JTAG k nahrání konfigurace.
- řada mikrokontrolérů používá JTAG jako ladicí rozhraní, někdy ovšem počet pinů nedovoluje JTAG použít a proto se můžeme setkat i s jednovodičovými proprietárními rozhraními
- procesory MIPS a PowerPC podporují JTAG
- dříve používal Intel JTAG jako ladicí rozhraní, dnes už pravděpodobně slouží pouze jako *Boundary Scan*, tedy k testování číslicového obvodu.
- sběrnice jako např. PCI nebo PCI-Express mají část signálů vyhrazeny pro JTAG.

2.4 Výhody a nevýhody

Mezi jasné výhody rozhraní JTAG je možné zařadit především velmi malý počet povinných signálů – každý výrobce procesoru se snaží co nejlépe využít piny svého obvodu a JTAG mu tak vychází vstříc. Další velkou výhodou je samotná standardizace a s tím související rozšíření mezi téměř všemi výrobci.

Naopak nevýhodou může být poměrně malá propustnost a z hlediska použití jako ladicí rozhraní pak také charakter komunikace, která je iniciovaná vždy testovacím nástrojem generujícím hodinový signál `TCK`. Proto když chceme čekat na nějakou událost uvnitř testovaného obvodu, nezbývá nám než pravidelně vyčítat obsah nějakého stavového registru (tzv. *polling*).

Kapitola 3

Standard Nexus 5001 (IEEE-ISTO 5001™-2003)

Nexus 5001 Forum™ bylo založeno za účelem vývoje univerzálního ladicího rozhraní vestavěných systémů, jejichž výkon a složitost neustále rostou a je potřeba hledat efektivní nástroje, pomocí nichž lze vyvíjené aplikace ladit, případně sledovat určité parametry za běhu aplikace.

Absence standardu v této oblasti brzdila vývojáře ladicích nástrojů a integrovaných vývojových prostředí, neboť každý výrobce disponoval svým vlastním proprietárním řešením, o kterém často ani nebyl dostatek informací.

Aby bylo zajištěno rozšíření tohoto standardního rozhraní, bylo velmi rozumné vzít v úvahu existující a již velmi rozšířené rozhraní a protokol JTAG, jenž dnes najdeme téměř v každém zařízení. Vzhledem k nízké propustnosti a nemožnosti asynchronních událostí iniciovaných samotným zařízením byl navíc definován rozšiřující port, tzv. *Auxiliary Port* (AUX), který je možné použít souběžně s rozhraním JTAG a nebo zcela samostatně. Standard Nexus 5001 pak definuje funkci těchto rozšiřujících pinů, protokol přenosu dat a standardní prostředky pro ladění aplikace na vestavěném zařízení [1].

Tyto prostředky lze rozdělit do dvou skupin – kontrola a analýza běhu aplikace (nebo též statické a dynamické funkce). První z nich zahrnuje čtení a modifikaci vnitřních registrů a pamětí, když je procesor zastaven, ruční zastavení/spuštění běhu procesoru, nastavení breakpointů a watchpointů a krokování. Druhá skupina pak zahrnuje především analytické nástroje používané za běhu aplikace nemajíce výrazný vliv na její běh. Mezi ně lze zařadit *Ownership Trace*, *Program Trace* nebo *Data Trace*. Jejich význam bude vysvětlen v následujících kapitolách.

3.1 Definice základních pojmů

V této kapitole uvedu řadu pojmů, které jsou ve standardu definovány a budou se v práci dále hojně vyskytovat [1]:

User Mode Mód, kdy procesor běží, jsou dostupné pouze dynamické funkce.

Debug Mode Mód, kdy je procesor zastaven a funkce pro statické ladění jsou aktivovány.

Breakpoint Procesor je zastaven, pokud je splněna nastavená podmínka.

Data Breakpoint Breakpoint, který je vyvolán při události čtení resp. zápisu na určitou adresu nebo při přečtení resp. zapsání určité hodnoty.

Instruction Breakpoint Breakpoint, který je vyvolán, pokud je adresa v programovém čítači shodná s nastavenou adresou.

Watchpoint Breakpoint, který nevyvolá zastavení procesoru, ale pouze aktivuje stavový signál, který je přes ladicí rozhraní detekován.

Ownership Trace Monitorování vlastnictví procesu (předpokládá operační systém)

Program Trace Monitorování toku programu

Data Trace Monitorování zapisování dat

Auxiliary Port (AUX) Rozšiřující port složený ze samostatných vstupních (AUX IN) a výstupních (AUX OUT) signálů.

Klient (Client) Funkční blok, který je zpřístupněn ladicím rozhraním – typicky jádro procesoru.

Nexus Interní název pro tento standard.

Public Messages Zprávy definované v protokolu pro rozhraní JTAG i AUX.

Nexus-Recommended Register (NRR) Kontrolní, stavové a datové registry doporučené standardem pro určitou množinu funkčnosti.

3.2 Klasifikace podle funkčnosti a výkonu

Standard Nexus 5001 definuje velmi škálovatelné rozhraní – výrobce vestavěného zařízení si může zvolit, kolik funkčnosti vývojářům poskytne a tím ovlivnit i cenu implementace ladicího rozhraní. Tento nástroj totiž pochopitelně zabírá plochu a piny na čipu a tím cenu zařízení ovlivňuje. Základní množina funkčností je přizpůsobena rozhraní JTAG a poskytuje tedy pouze nejnutnější prostředky statického vývoje. Při vyšších požadavcích je nutné použít rozšiřujícího AUX portu.

Z hlediska množiny funkcí rozlišuje Nexus 4 třídy. První třída odpovídá minimu funkčnosti za použití pouze rozhraní JTAG, poslední, čtvrtá, třída pak poskytuje pokročilé možnosti ladění, které jsou již často spjaty s operačním systémem běžícím ve vestavěném systému. V tabulkách 3.1 a 3.2 jsou uvedeny funkce, které musí zařízení splňující jednotlivé třídy implementovat (V = definované výrobcem, R = povinné, O = volitelné) [1].

Druhým kritériem, podle kterého je možné rozhraní klasifikovat, je výkon. Jestliže zařízení implementuje pouze rozhraní JTAG, komunikace je pouze poloviční duplexní (*Half-duplex*). Při požadavku vyšší než první třídy je nutné implementovat AUX port a komunikace je již plně duplexní (*Full-duplex*). V závislosti na výkonnosti zařízení a požadovaných schopnostech ladicího rozhraní je možné rozšiřující AUX port škálovat upravením počtu AUX IN a AUX OUT portů.

Funkce	Třída 1	Třída 2	Třída 3	Třída 4
Čtení/zápis uživatelských registrů v debug módu	V	V	V	V
Čtení/zápis uživatelské paměti v debug módu	R	R	R	R
Vstup do debug módu po resetu procesoru	R	R	R	R
Vstup do debug módu z uživatelského módu	R	R	R	R
Opuštění debug módu do uživatelského módu	R	R	R	R
Krokování instrukcí v uživatelském módu a přechod zpět do debug módu	R	R	R	R
Zastavení vykonávání programu na instrukčním/-datovém breakpointu a vstup do debug módu (minimálně 2 breakpointy)	R	R	R	R

Tabulka 3.1: Klasifikace funkcí pro statické ladění [1].

Funkce	Třída 1	Třída 2	Třída 3	Třída 4
Možnost nastavit breakpoint nebo watchpoint	R	R	R	R
Identifikace zařízení	R	R	R	R
Možnost zaslat událost ze zařízení, když nastane watchpoint	O[1]	R	R	R
Monitorování vlastnictví procesu za běhu procesoru v reálném čase	-	R	R	R
Monitorování toku programu za běhu procesoru v reálném čase	-	R	R	R
Monitorování zapisování dat za běhu procesoru v reálném čase	-	-	R	R
Čtení/zápis uživatelské paměti za běhu procesoru v reálném čase	-	-	R	R
Spouštění programu z Nexus portu	-	-	-	R
Možnost spustit monitorování vlastnictví procesu, toku programu nebo zapisování dat po výskytu watchpointu	-	-	-	R
Možnost spustit nahrazení obsahu paměti po výskytu watchpointu nebo přístupu programu na definovanou adresu	-	-	-	O
Monitorování čtení uživatelské paměti za běhu procesoru v reálném čase	-	-	O	O

Tabulka 3.2: Klasifikace funkcí pro dynamické ladění [1].

3.3 Rozhraní

Rozhraní ladicího nástroje podle Nexus 5001 se liší podle zvolené třídy funkčnosti – zařízení podporující třídu 1 mohou implementovat pouze rozhraní JTAG, zařízení vyšších tříd musí implementovat AUX port a volitelně JTAG. Komunikace přes AUX port je výhradně paketově orientovaná (tzv. *Public Messages*), v případě JTAGu je možné komunikovat jak pomocí paketů, tak pomocí definovaného protokolu. Vzhledem k tomu, že v mé práci se zabývám pouze implementací třídy 1 a pro komunikaci využívám pouze JTAG, nebudu se paketové komunikaci dále věnovat.

V tabulce 3.3 jsou uvedeny všechny signály tvořící rozhraní ladicího nástroje. U každého

[1] V třídě 1 je možné implementovat pomocí signálu $\overline{\texttt{EVTO}}$.

z nich je uveden směr komunikace a jestli jsou povinné nebo volitelné pro jednotlivé výkonové třídy. V posledním sloupci je pak uveden význam signálu. Je vidět, že k implementaci základní funkčnosti ladicího rozhraní opravdu stačí 4 signály JTAGu, volitelně doplněné o asynchronní reset TRST a 3 další asynchronní signály.

Signál	Směr	Full-duplex (AUX IN/AUX OUT)	Full-duplex (JTAG)	Half-duplex (JTAG)	Význam
MCKI	Vstup	R	-	-	*Message Clock In*, vstupní hodinový signál pro časování MDI a MSEI.
RSTI	Vstup	R	-	-	Asynchronní reset Nexus portu.
MDI[N:0]	Vstup	R	-	-	*Message Data In*, vstupní datový signál AUX portu.
MSEI[1:0]	Vstup	R	-	-	*Message Start/End In*, vstupní signalizace začátku a konce paketu a zprávy.
MCKO	Výstup	R	R	-	*Message Clock Out*, výstupní hodinový signál pro časování MDO a MSEO.
MDO[M:0]	Výstup	R	R	-	*Message Data Out*, výstupní datový signál AUX portu.
MSEO[1:0]	Výstup	R	R	-	*Message Start/End Out*, výstupní signalizace začátku a konce paketu a zprávy.
EVTI	Vstup	R	O	O	*Event In*, sestupná hrana způsobí zastavení procesoru nebo zaslání zprávy o *Program Trace* nebo *Data Ttrace*.
EVTO	Výstup	O	O	O	*Event Out*, signalizuje výskyt breakpointu.
TCK	Vstup	-	R	R	Hodinový signál JTAGu (viz kapitola 2.1).
TMS	Vstup	-	R	R	Řídicí signál JTAGu (viz kapitola 2.1).
TDI	Vstup	-	R	R	Vstupní datový signál JTAGu (viz kapitola 2.1).
TDO	Výstup	-	R	R	Výstupní datový signál JTAGu (viz kapitola 2.1).
TRST	Vstup	-	O	O	Asynchronní reset JTAGu (viz kapitola 2.1).
RDY	Výstup	-	O	O	*Ready*, signalizuje připravená data a tím zrychluje komunikaci přes rozhraní JTAG.

Tabulka 3.3: Signály rozhraní Nexus 5001 [1]

Šířka signálů MDI[N:0] a MDOM:0 závisí na třídě funkčností a na výkonnosti vestavěného zařízení, pro každou třídu je pouze specifikována minimální a maximální doporučená šířka (zvlášť pro vstupní a výstupní signály) [1]. Tato šířka se pohybuje od 1 do 16, přičemž větší šířky jsou typické zejména pro výstupní datové signály, neboť je potřeba přenášet velké množství dat při monitorování toku programu, dat nebo vlastnictví procesu.

3.4 Doporučené registry Nexus portu

Standard Nexus 5001 doporučuje některé stavové, kontrolní a datové registry pro jednotlivé třídy funkčnosti. Tyto registry jsou přístupné buď paketově pomocí *Nexus Public Messages* nebo pomocí definovaného protokolu přes rozhraní JTAG (bude popsáno dále). Zápisem do těchto registrů je možné ovládat funkce pro statické i dynamické ladění. Úplný seznam doporučených registrů je možné najít v [1], pro účely této práce zde uvedu pouze registry, jejichž význam je pro následující text nezbytný. Každý registr má své číslo, pomocí kterého je k němu poté přistupováno, může z něj být čteno nebo do něj zapisováno. Seznam vybraných registrů je uveden v tabulce 3.4, detailnější popis následuje v dalších podkapitolách.

Registr	**Třída**	**Číslo registru**	**Čtení/Zápis**
Device ID (DID)	všechny	0	R
Client Select Control (CSC)	2, 3, 4	1	R/W
Development Control (DC)	2, 3, 4	2	R/W
Development Status (DS)	2, 3, 4	2	R
Read/Write Access Control/Status (RWCS)	3, 4	7	R/W
Read/Write Access Address (RWA)	3, 4	9	R/W
Read/Write Access Data (RWD)	3, 4	10	R/W
Breakpoint/Watchpoint Control (BWC) (2)	4	22-23	R/W
Breakpoint/Watchpoint Address (BWA) (2)	4	30-31	R/W
Breakpoint/Watchpoint Data (BWD) (2)	4	38-39	R/W
Definováno výrobcem	-	64-127	-

Tabulka 3.4: Doporučené registry Nexus portu

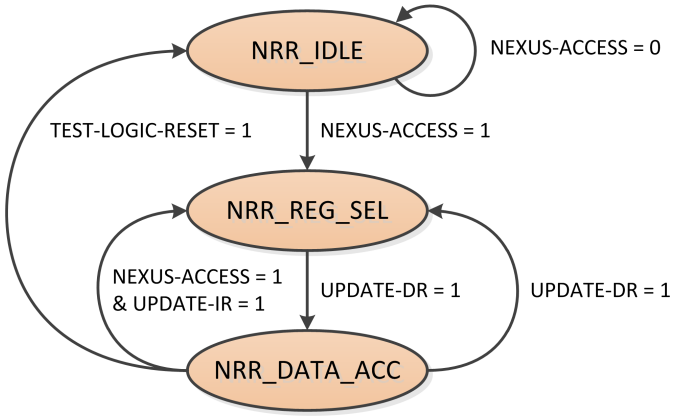
Registry, které jsem v této tabulce neuvedl, se týkají především funkcí pro dynamické ladění, které nejsou předmětem této práce. Registr DID je sice povinný pro všechny třídy, ale pokud je součástí Nexus portu rozhraní JTAG, obsahuje již identifikační registr (viz kapitola 2.1.4), takže není tento registr potřeba duplikovat.

3.4.1 Protokol pro přístup k registrům

Pokud zařízení implementuje pouze první třídu funkčnosti pomocí rozhraní JTAG, standard Nexus 5001 definuje protokol, pomocí kterého je možné tyto registry číst a zapisovat do nich. Aktivace komunikace pomocí tohoto protokolu je provedena načtením speciální instrukce NEXUS-ACCESS do instrukčního registru JTAG TAP. Samotná komunikace je pak založena na 3 stavech - nečinný (*NRR_IDLE*), výběr registru (*NRR_REG_SEL*) a přístup k registru (*NRR_DATA_ACC*). Stavy a přechody mezi nimi jsou zobrazeny na obrázku 3.1.

Řadič Nexus portu se nachází po resetu ve stavu *NRR_IDLE*. Po nahrání instrukce NEXUS-ACCESS se přesune do stavu *NRR_REG_SEL* a očekává datový scan, kterým je nahráno číslo registru a režim přístupu do osmibitového registru NRRS (Nexus Recommended Register Select – viz obrázek 3.2).

Po zapsání hodnoty do tohoto registru – 7bitové adresy registru a 1bitového příznaku, jestli chceme číst (nula) nebo zapisovat (jednička) – je řadič ve stavu *NRR_DATA_ACC*. Nyní můžeme datovým scanem přistupovat k obsahu vybraného registru. Po ukončení datového scanu je řadič opět ve stavu *NRR_IDLE*. Příklad takové komunikace je uveden v tabulce 3.5.



Obrázek 3.1: Stavový automat protokolu pro přístup k registrům [1]

Obrázek 3.2: *Nexus Recommended Register Select* [1]

Krok	TMS	**Stav JTAGu**	**Stav Nexusu**	**Popis**
1	0	*Run-Test/Idle*	*NRR_IDLE*	JTAG je po resetu.
2	1	*Select-DR-Scan*	*NRR_IDLE*	
3	1	*Select-IR-Scan*	*NRR_IDLE*	
4	0	*Capture-IR*	*NRR_IDLE*	
5	0	*Shift-IR*	*NRR_IDLE*	Začíná přenos instrukce NEXUS-ACCESS.
M-1 period TCK				Instrukce NEXUS-ACCESS je přenášena.
6	1	*Exit1-IR*	*NRR_IDLE*	
7	1	*Update-IR*	*NRR_IDLE*	NEXUS-ACCESS je zapsána do instr. reg.
8	0	*Run-Test/Idle*	*NRR_REG_SEL*	Nexus je připraven přijmout číslo registru.
9	1	*Select-DR-Scan*	*NRR_REG_SEL*	
10	0	*Capture-DR*	*NRR_REG_SEL*	
11	0	*Shift-DR*	*NRR_REG_SEL*	Začíná přenos čísla registru.
7 period TCK				Probíhá přenos čísla registru.
12	1	*Exit1-DR*	*NRR_REG_SEL*	
13	1	*Update-DR*	*NRR_REG_SEL*	Číslo registru je zapsáno do NRRS.
14	1	*Select-DR-Scan*	*NRR_DATA_ACC*	Nexus je připraven přenášet data.
14	0	*Capture-DR*	*NRR_DATA_ACC*	
14	0	*Shift-DR*	*NRR_DATA_ACC*	Začíná přenos dat.
N-1 period TCK				Probíhá přenos dat.
14	1	*Exit1-DR*	*NRR_DATA_ACC*	
14	1	*Update-DR*	*NRR_DATA_ACC*	Data jsou zapsána do vybraného registru.
14	0	*Run-Test-Idle*	*NRR_IDLE*	Přenos je ukončen.

Tabulka 3.5: Příklad sekvence pro zápis do registru [1]

3.4.2 Client Select Control (CSC)

Tento registr je doporučen v případě, že vestavěné zařízení obsahuje více klientů. Nahráním 5bitového čísla klienta volíme, ke kterému klientovi bude přistupováno pomocí protokolu představeného v předchozí kapitole. Zbylé tři bity jsou rezervovány pro budoucí použití.

Obrázek 3.3: Registr *Client Select Control* (CSC)

3.4.3 Development Control (DC)

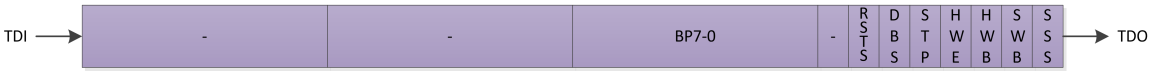
Obrázek 3.4: Registr *Development Control* (DC)

Registr *Development Control* (DC) je určen k základnímu ovládání klienta – bit 13 DBE (*Debug Enable*) povoluje debug mód, bit 12 DBR (*Debug Request*) umožňuje ručně procesor zastavit nebo opět spustit. Pokud je debug mód povolený, může dojít k zastavení procesoru při nastavení DBR nebo breakpointu. Povolení debug módu je také nezbytné pro krokování, které je realizováno bitem 8 SS (*Step Enable*).

Bit 14 CBI (*Client Breakpoint Input*) způsobí u zařízení s více klienty, že bude klient zastaven nejen na vlastní breakpoint, ale také na tzv. globální breakpoint, tedy na breakpoint libovolného jiného klienta.

Bity 31-24 jsou definované výrobcem zařízení, bity 23-15 jsou rezervovány pro pozdější použití. Ostatní bity nejsou pro tuto práci podstatné, jejich význam lze nalézt v standardu [1].

3.4.4 Development Status (DS)



Obrázek 3.5: Registr *Development Status* (DS)

Přečtením registru *Development Status* (DS) získáme informaci, v jakém stavu se nachází klient. Bity 15-8 BP7-0 (*Breakpoint Status*) signalizují, jestli došlo k nějakému breakpointu. Bit 5 DBS (*Debug Status*) informuje o stavu procesoru – pokud je roven jedné, procesor je zastaven v debug módu, jinak běží. Bity 2 HWB (*Hardware Breakpoint Status*) a 1 SWB (*Software Breakpoint Status*) indikují, zda došlo k hardwarovému resp. softwarovému breakpointu. Poslední bit 0 SSS (*Single Step Status*) je aktivní, pokud byl procesor zastaven po krokování.

Bity 31-24 jsou definované výrobcem, bity 23-16 a 7 jsou rezervované pro budoucí použití. Ostatní bity nejsou důležité, jejich význam je opět uveden v standardu [1].

3.4.5 Read/Write Access Control/Status (RWCS)

Obrázek 3.6: Registr *Read/Write Access Control/Status* (RWCS)

Pro čtení a zápis do uživatelské paměti slouží registry RWCS, RWA a RWD. Registr RWCS slouží zároveň jako kontrolní i stavový registr. Je do něj tedy zapisováno jak ze strany ladicího nástroje, tak přes Nexus port. Nastavením bitu 31 AC (*Access Control*) lze zahájit nebo ukončit přenos. Bit 30 RW (*Read/Write*) určuje, zda se bude z paměti číst, nebo se do ní bude zapisovat. Bity 29-27 SZ (*Word Size*) jsou pevně nastavené a určují velikost slova v paměti. Bity 26-24 MAP (*Map Select*) umožňují přepínat přístup do více pamětí, pokud jsou k dispozici. Bity 15-2 CNT (*Access Count*) umožňují podporu blokového čtení a zápisu. Bit 1 ERR (*Error*) je stavový a indikuje chybu v přístupu do paměti. Poslední bit 0 DV (*Data Valid*) oznamuje, že jsou v registru BWD platná data.

Při blokovém čtení se postupuje podle následujících bodů:

1. Nahrání počáteční adresy do registru RWA.
2. Nastavení RWCS: AC = 1, RW = 0, CNT = počet slov, které chceme přečíst minus jedna (čte se blok paměti od RWA do RWA+CNT).
3. Čekání na DV = 1, poté čtení registru RWD.
4. Pokud je CNT = 0, přenos končí, jinak pokračuje bodem 3.

Pro blokový zápis je potřeba postupovat podle následujících kroků:

1. Nahrání počáteční adresy do registru RWA.
2. Zápis dat do registru RWD.
3. Nastavení RWCS: AC = 1, RW = 1, DV = 1, CNT = počet slov, které chceme zapsat minus jedna (zapisuje se blok paměti od RWA do RWA+CNT).
4. Čekání na DV = 0, poté pokud je CNT = 0, přenos končí, jinak zápis do registru BWD a opakování kroku.

3.4.6 Read/Write Access Address (RWA) a Data (RWD)

Registry RWA a RWD mají výrobcem definovanou šířku a slouží společně s registrem RWCS k přístupu do uživatelské paměti.

3.4.7 Breakpoint/Watchpoint Control (BWC)

Registrů BWC, BWA a BWD může být v zařízení 2-8 podle počtu implementovaných breakpointů. Struktura registru BWC je zobrazena na obrázku 3.7.

Bity 31-30 BWE (*Breakpoint/Watchpoint Enable*) povolují breakpoint resp. watchpoint – při hodnotě 00 je breakpoint i watchpoint zakázán, 01 značí povolený breakpoint a 11 povolený watchpoint, hodnota 10 je rezervována.

Bity 29-28 BRW (*Breakpoint/Watchpoint Read/Write Select*) rozhodují, při jaké události bude vyvolán breakpoint resp. watchpoint. Pokud jsou nastaveny na hodnotu 00, k breakpointu dojde při čtecím přístupu do paměti, při hodnotě 01 při zápisu do paměti, při hodnotě 11 při libovolném přístupu do paměti a hodnota 11 je rezervována.

Bity 17-16 BWO (*Breakpoint/Watchpoint Operand*) určují, s kterým registrem bude hodnota porovnávána, jestliže je nastaven bit 17, pak s BWA, jestliže bit 16, pak s BWD. Pokud jsou nastaveny oba bity, bude se porovnávat s oběma registry.

Bitem 15 BWT (*Breakpoint/Watchpoint Type*) se dá ovlivnit, jestli se jedná o instrukční nebo datový breakpoint.

Obrázek 3.7: Registr *Breakpoint/Watchpoint Control* (BWC)

3.4.8 Breakpoint/Watchpoint Address (BWA) a Data (BWD)

Registry BWA a BWD jsou podobně jako registry RWA a RWD závislé na šířce adresové a datové sběrnice procesoru. Jejich počet odpovídá počtu implementovaných breakpointů, přičemž minimální počet jsou 2 a maximální 8.

3.5 Podpora ze strany výrobců

Rozšířenost standardu Nexus 5001 je zatím poměrně nízká, což je dáno především tím, že je poměrně nový. Druhým důvodem ale může být i poměrně složitá implementace vyšších tříd funkčnosti. K výrobcům, kteří Nexus implementují do svých zařízení, patří především Freescale Semiconductor, v jehož mikrokontrolérech můžeme najít Nexus port třídy 2 a vyšší.

Kapitola 4

Implementace nástroje

Po spíše teoretických úvodních kapitolách bude obsahem této kapitoly popis konkrétní implementace ladicího rozhraní podle standardu Nexus 5001. Navržený nástroj splňuje první třídu funkčnosti, ke komunikaci využívá pouze rozhraní JTAG, které je možné jednoduše doplnit několika asynchronními signály. Jelikož je součástí celkového návrhu ladicího nástroje rozhraní JTAG, zaměřím se nejprve na detailní popis implementace TAP.

4.1 Návrh JTAG Test Access Portu

Při návrhu rozhraní podle standardu JTAG jsem postupoval dekompozicí celého obvodu na menší celky s jasně definovaným chováním a rozhraním. Tyto komponenty jsem následně popsal v jazyku VHDL. Entitou nejvyšší úrovně (`tap`) byl samotný *Test Access Port*, ten jsem rozdělil na komponenty přesně podle obrázku 2.2, vytvořil jsem tedy komponenty pro řadič TAP `tap_ctrl`, instrukční registr `instr_reg`, instrukční dekodér `instr_dec`, bypass registr `bypass_reg`, identifikační registr `idcode_reg` a testovací řetězec buněk `scan_chain`.

V následujících podkapitolách nejprve popíšu jednotlivé komponenty, jejich rozhraní a implementaci, a nakonec uvedu jejich zapojení do TAP. Pro zvýšení přehlednosti kódu jsem vytvořil balík `jtag`, ve kterém jsou definovány konstanty společné pro celý TAP.

4.1.1 Řadič TAP

Rozhraní entity `tap_ctrl` tvoří 3 vstupní signály – `TCK`, `TMS` a $\overline{\texttt{TRST}}$ – signály rozhraní JTAG. Výstupní signály jsou řídicí signály vedoucí k registrům a multiplexorům.

```
entity tap_ctrl is
    port (
        TCK:       in  std_logic;
        TMS:       in  std_logic;
        TRST:      in  std_logic;
        EnableDR:  out std_logic;
        ShiftDR:   out std_logic;
        UpdateDR:  out std_logic;
        SelectR:   out std_logic;
        ShiftIR:   out std_logic;
        UpdateIR:  out std_logic;
        EnableIR:  out std_logic;
        Reset:     out std_logic;
        Enable:    out std_logic
    );
end tap_ctrl;
```

Architektura této entity je popsána behaviorálně pomocí dvou procesů a několika podmíněných přiřazení. Celý řadič funguje jako konečný stavový automat s 16 stavy (viz kapitola 2.1.1).

```
architecture tap_ctrl_behav of tap_ctrl is
    type state_t is (StTestLogicReset, StRunTestIdle, StSelectDR, StSelectIR,
                     StCaptureDR, StShiftDR, StExit1DR, StPauseDR, StExit2DR,
                     StUpdateDR, StCaptureIR, StShiftIR, StExit1IR,
                     StPauseIR, StExit2IR, StUpdateIR);
    signal PresentState, NextState: state_t;
begin
```

Prvním procesem Moorova konečného automatu je proces `present_state_reg`. Tento proces je citlivý na změnu signálů `TCK` a $\overline{\texttt{TRST}}$ a představuje registr uchovávající aktuální stav automatu. Na náběžnou hranu hodin `TCK` je do registru zapsána hodnota `NextState`, při aktivním asynchronním resetu $\overline{\texttt{TRST}}$ je do registru zapsán stav *Test-Logic-Reset*.

```
    present_state_reg: process (TCK, TRST)
    begin
        if (TRST = '0') then
            PresentState <= StTestLogicReset;
        elsif (rising_edge(TCK)) then
            PresentState <= NextState;
        end if;
    end process;
```

Druhý proces `next_state_logic` modeluje kombinační logiku příštího stavu konečného automatu. Je citlivý na změnu signálů `PresentState` a `TMS`.

```
    next_state_logic: process(PresentState, TMS)
    begin
        case PresentState is
            when StTestLogicReset =>
                case TMS is
                    when '1'    => NextState <= StTestLogicReset;
                    when others => NextState <= StRunTestIdle;
                end case;
            when StRunTestIdle =>
                case TMS is
                    when '1'    => NextState <= StSelectDR;
                    when others => NextState <= StRunTestIdle;
                end case;
            when StSelectDR =>
                case TMS is
                    when '1'    => NextState <= StSelectIR;
                    when others => NextState <= StCaptureDR;
                end case;

            ...

        end case;
    end process;
```

Zbytek architektury entity tvoří výstupní logika stavového automatu. Místo signálů `ClockDR` a `ClockIR` jsou zde signály `EnableDR` a `EnableIR`, tím se řeší problém hradlovaných hodin, což je praktika, které je dobré se vyhnout. Pokud totiž na hodinový vstup registru přivedeme kombinační signál, hrozí, že bude docházet k hazardům a obvod nebude fungovat tak, jak má. Hradlování hodin není doporučené zejména pokud je cílovou platformou FPGA, v ASIC obvodech se s hradlováním můžeme setkat běžněji. Jiným řešením by mohlo být použití jednotek DCM, které jsou přítomny na FPGA. Těch je ale omezené množství a celý obvod by se navíc stával platformě závislý.

```
-- Output Logic
EnableIR <= '1' when (PresentState = StCaptureIR or
                      PresentState = StShiftIR) else '0';
ShiftIR  <= '1' when (PresentState = StShiftIR) else '0';
UpdateIR <= '1' when (PresentState = StUpdateIR) else '0';
EnableDR <= '1' when (PresentState = StCaptureDR or
                      PresentState = StShiftDR or
                      PresentState = StUpdateDR) else '0';
ShiftDR  <= '1' when (PresentState = StShiftDR) else '0';
UpdateDR <= '1' when (PresentState = StUpdateDR) else '0';
SelectR  <= '1' when (PresentState = StCaptureIR or
                      PresentState = StShiftIR or
                      PresentState = StExit1IR or
                      PresentState = StPauseIR or
                      PresentState = StExit2IR or
                      PresentState = StUpdateIR) else '0';
Enable   <= '1' when (PresentState = StShiftDR or
                      PresentState = StShiftIR) else '0';
Reset    <= '1' when (PresentState = StTestLogicReset or
                      TRST = '0') else '0';
```

4.1.2 Instrukční registr

Rozhraní instrukčního registru tvoří hodinový signál `CLK`, asynchronní reset `RST`, povolovací vstup `E`, dále pak sériový vstup `SI`, řídicí signály `Shift` a `Update` vedoucí z řadiče TAP, sériový výstup `SO` a paralelní výstup `PO`.

Chování této komponenty je tvořeno dvěma procesy. První z nich tvoří posuvný registr zapojený mezi sériový vstup a výstup a posouvaný na náběžnou hranu hodin `CLK` při aktivním povolovacím vstupu `E` a aktivním signálu `Shift`. Pokud není aktivní signál `Shift`, do posuvného registru je načtena hodnota 0001, první dva bity však mohou být ovlivněny libovolnými stavovými signály (toho bude dále využito). Tato hodnota je pak vysouvána na výstup `TDO` celého rozhraní.

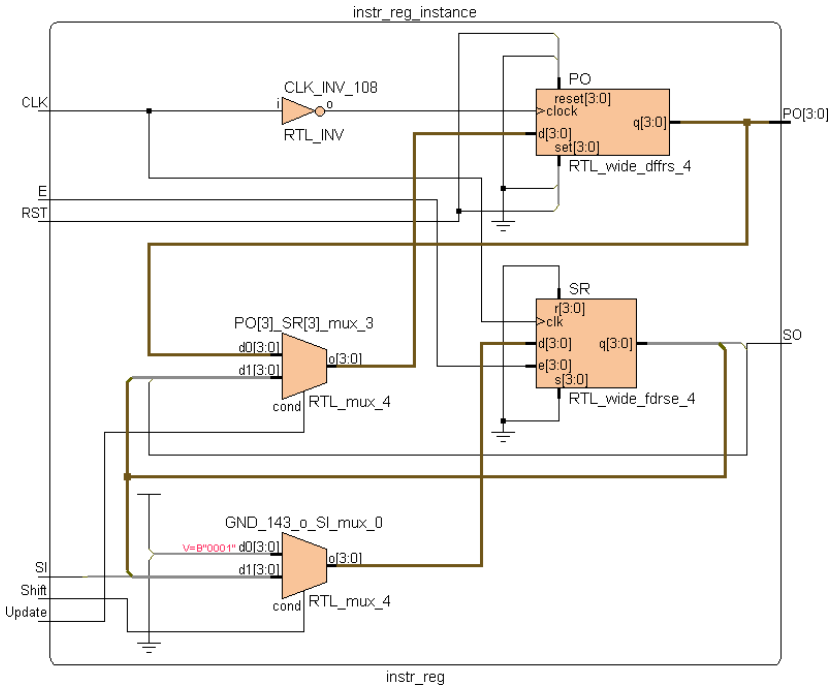
```
isr: process (CLK)
begin
    if (rising_edge(CLK)) then
        if (E = '1') then
            if (Shift = '1') then
                SR <= SI & SR(IR_SIZE-1 downto 1);
            else
                SR <= (0 => '1', 1 => '0', others => '0');
            end if;
        end if;
    end if;
end process;
```

Druhý proces modeluje samotný instrukční registr, data z posuvného registru jsou do instrukčního registru zapisována na sestupnou hranu hodinového signálu při aktivním řídicím signálu `Update`. Registr také podporuje asynchronní reset, při kterém je do registru načtena výchozí instrukce (zde IDCODE, ale pokud TAP neobsahuje identifikační registr, pak je výchozí instrukcí vždy BYPASS).

```
isl: process (CLK, RST)
begin
    if (RST = '1') then
        PO <= I_DEFAULT;
    elsif (falling_edge(CLK)) then
        if (Update = '1') then
            PO <= SR;
```

```
            end if;
        end if;
    end process;
```

Schématický pohled na instrukční registr je vidět na obrázku 4.1, který byl vytvořen v programu PlanAhead firmy Xilinx.



Obrázek 4.1: Instrukční registr

4.1.3 Instrukční dekodér

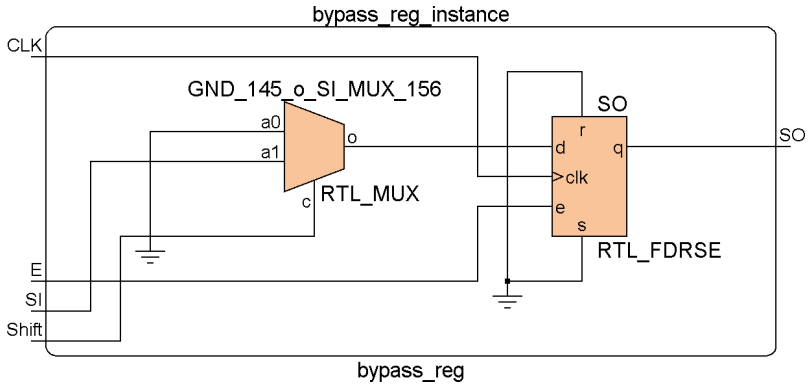
Instrukční dekodér je čistě kombinatorický obvod, jehož vstupem je pouze instrukce z instrukčního registru a výstupem je řídicí signál `MX1Select` vedoucí k výstupnímu multiplexoru, a povolovací signály vedoucí k bypass registru, identifikačnímu registru a testovacímu řetězci. Posledním výstupem je signál `Mode`, který ovládá režim testovacích buněk.

```
architecture instr_dec_behav of instr_dec is
begin
    MX1Select <= MX1_TDI        when (Instruction = I_CLAMP) else
                 MX1_SCAN_CHAIN when (Instruction = I_EXTEST or
                                     Instruction = I_SAMPLE) else
                 MX1_IDCODE     when (Instruction = I_IDCODE) else
                 MX1_BYPASS;

    BypassRegEnable <= '1' when (Instruction = I_BYPASS) else '0';
    IdcodeRegEnable <= '1' when (Instruction = I_IDCODE) else '0';
    ScanChainEnable <= '1' when (Instruction = I_EXTEST or
                                 Instruction = I_SAMPLE) else '0';

    Mode <= '0' when (Instruction = I_BYPASS or
                      Instruction = I_IDCODE or
                      Instruction = I_SAMPLE) else '1';
end instr_dec_behav;
```

4.1.4 Bypass registr

Tento registr slouží ke zkrácení testovacího řetězce, pokud je do něj zapojeno více TAP. Jedná se o jednobitový registr, na jeho výstup je na náběžnou hranu hodin zapisována hodnota ze vstupu, pokud je aktivní signál `Shift`, jinak je na výstup zapsána hodnota 0. Tím lze automaticky rozlišit všechny TAP zapojené do jednoho řetězce (viz kapitola 2.1.4).

```
architecture bypass_reg_behav of bypass_reg is
begin
    bypass: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (E = '1') then
                if (Shift = '1') then
                    SO <= SI;
                else
                    SO <= '0';
                end if;
            end if;
        end if;
    end process;
end bypass_reg_behav;
```



Obrázek 4.2: Bypass registr

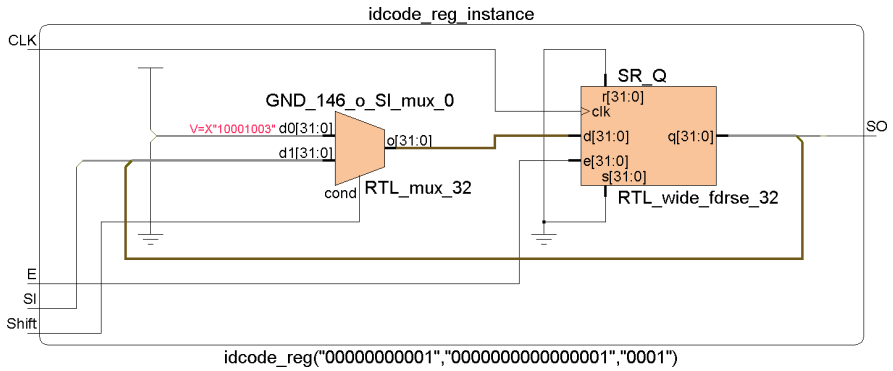
4.1.5 Identifikační registr

Identifikační registr má šířku 32 bitů a jeho rozhraní tvoří generické hodnoty (identifikace výrobce, číslo a verze výrobku) a vstupní a výstupní signály (hodinový signál `CLK`, povolovací vstup `E`, sériový vstup `SI`, řídicí signál `Shift` a sériový výstup `SO`). Generické parametry jsou spojeny do konstanty `IDCODE`, která tvoří celý identifikační řetězec. Ten je načten do posuvného registru a sériově vysunut přes `TDO`. Nejméně významný bit je vždy roven 1 (viz kapitola 2.1.4).

Chování této komponenty je popsáno jedním procesem modelujícím samotný registr a podmíněným přiřazením reprezentujícím vstupní multiplexor (ten přepíná režim posuvu a nahrání konstantního identifikačního řetězce). Schéma komponenty je vidět na obrázku 4.3.

```
architecture idcode_reg_behav of idcode_reg is
    constant IDCODE: std_logic_vector(31 downto 0)
        := Version & PartNumber & Manufacturer & '1';
```

```
    signal SR_D: std_logic_vector(31 downto 0);
    signal SR_Q: std_logic_vector(31 downto 0);
begin
    SR_D <= SI & SR_Q(31 downto 1) when (Shift = '1') else IDCODE;

    idcode_reg: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (E = '1') then
                SR_Q <= SR_D;
            end if;
        end if;
    end process;

    SO <= SR_Q(0);
end idcode_reg_behav;
```



Obrázek 4.3: Identifikační registr

4.1.6 Testovací řetězec

Poslední částí TAP je testovací řetězec složený z *Boundary Scan Cell* buněk. Ty jsou sériově propojeny mezi sebou a jejich paralelní vstupy a výstupy jsou napojeny do datové cesty, která má být testována. Implementace jedné takové buňky je samostatnou entitou `scan_cell` s rozhraním mírně odlišným než na obrázku 2.7.

Oproti teoretickému schématu z obrázku 2.7 je v implementaci opět jedna změna – místo hodinového signálu `ClockDR`, který by musel být hradlován, je na každou buňku napojen přímo signál `TCK` a povolování je realizováno signálem `E` přivedeným na povolovací vstupy registrů. Každá *Boundary Scan Cell* buňka se skládá ze dvou multiplexorů a dvou registrů. Jejich přesná funkčnost byla popsána v kapitole 2.1.6.

```
architecture scan_cell_behav of scan_cell is
    signal CSC_D: std_logic;
    signal CSC_Q: std_logic := '0';
    signal UHC_D: std_logic;
    signal UHC_Q: std_logic := '0';
begin
    CSC_D <= PI when (Shift = '0') else
             SI;

    csc: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (E = '1' and Update = '0') then
```

```
                CSC_Q <= CSC_D;
            end if;
        end if;
    end process;

    SO    <= CSC_Q;
    UHC_D <= CSC_Q;

    uhc: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (E = '1' and Update = '1') then
                UHC_Q <= UHC_D;
            end if;
        end if;
    end process;

    PO <= PI when (Mode = '0') else
          UHC_Q;
end scan_cell_behav;
```

Spojením těchto buněk vznikne celý testovací řetězec `scan_chain`, jehož rozhraní obsahuje oproti samotné buňce paralelní vstup a výstup generické šířky `Size`.

Jednotlivé buňky jsou v řetězci instanciovány pomocí konstrukce *generate* a propojeny signály `ScanConn`. Dále jsou napojeny řídicí signály a paralelní vstupy a výstupy.

```
architecture scan_chain_behav of scan_chain is
    signal ScanConn: std_logic_vector(Size downto 0);
begin
    scan_cell_chain: for i in 1 to Size generate
        scan_cell_instance: entity work.scan_cell
            port map (
                CLK    => CLK,
                E      => E,
                SI     => ScanConn(i-1),
                PI     => PI(Size-i),
                Mode   => Mode,
                Shift  => Shift,
                Update => Update,
                SO     => ScanConn(i),
                PO     => PO(Size-i)
            );
    end generate;

    ScanConn(0) <= SI;
    SO          <= ScanConn(Size);
end scan_chain_behav;
```

4.1.7 Celkové zapojení

Všechny tyto komponenty jsou poté zapojeny do jednoho velkého obvodu `tap`, jehož rozhraním jsou signály JTAGu a datová cesta přerušená testovacím řetězcem. Šířka této datové cesty je dána generickým parametrem `DataSize`, hodnoty do identifikačního řetězce jsou také dány generickými parametry.

Architektura této entity pak kromě instancí všech popsaných komponent a jejich propojení obsahuje ještě dva multiplexory a výstupní třístavový budič.

```
    -- Multiplexer MX1
    MX1Out <= TDI          when (MX1Select = MX1_TDI) else
              ScanChainOut when (MX1Select = MX1_SCAN_CHAIN) else
              IdcodeRegOut when (MX1Select = MX1_IDCODE) else
              BypassRegOut;
```

```
-- Multiplexer MX2
MX2Out <= MX1Out        when (MX2Select = '0') else
          InstrRegOut;

-- Output Enable (tri-state)
oe: process (TCK)
begin
    if (falling_edge(TCK)) then
        if (Enable = '1') then
            TDO <= MX2Out;
        else
            TDO <= 'Z';
        end if;
    end if;
end process;
```

Pokud chceme celé rozhraní použít jako součást většího obvodu, stačí entitu `tap` instanciovat, signály `TCK`, `TMS`, `TDI`, `TDO`, případně $\overline{\texttt{TRST}}$ připojit na piny např. FPGA a signály `DataIn` a `DataOut` zapojit do datové cesty, kterou chceme testovat. Pokud volitelný asynchronní reset $\overline{\texttt{TRST}}$ nevyužijeme, nesmíme zapomenout připojit na tento signál konstantní hodnotu 1, protože je aktivní v nule.

4.1.8 Syntéza

Tabulka 4.1 ukazuje obsazenost logiky po syntéze samotného rozhraní JTAG bez žádného dalšího obvodu v programu Xilinx ISE. Je zde vidět srovnání obsazenosti FPGA Xilinx Spartan XC3S50-4PQ208C[1] při optimalizaci na rychlost a na plochu, rozdíl je evidentně velmi malý.

Typ logiky	Optimalizace na rychlost	Optimalizace na plochu
Slice Flip Flops	49	47
4vstupové LUT	64	64
Obsazené Slice	39	35

Tabulka 4.1: Obsazenost logiky po syntéze v programu Xilinx ISE

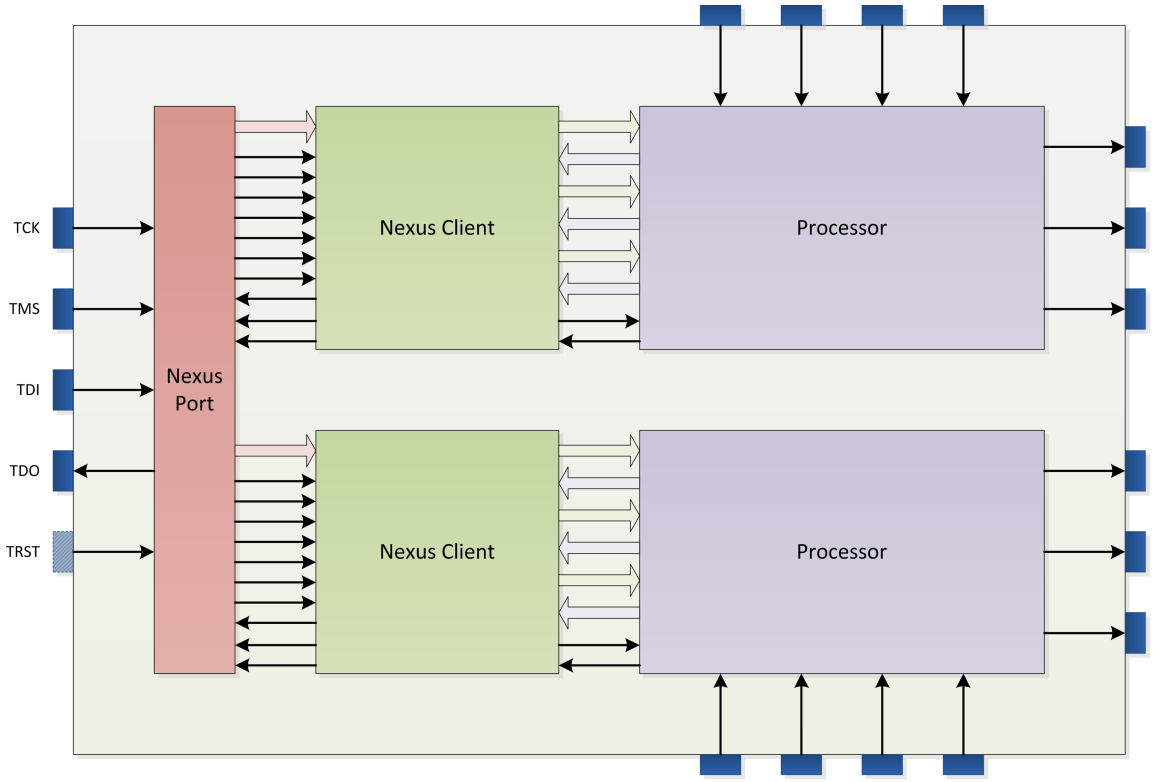
4.2 Návrh ladicího nástroje podle Nexus 5001

Po dokončení návrhu rozhraní JTAG a důkladném otestování jsem přistoupil k implementaci ladicího nástroje podle standardu Nexus 5001. Ten jsem nejprve rozdělil na dva velké celky:

- *Nexus Port* – V každém zařízení se nachází pouze jeden, obsahuje JTAG TAP, obsluhuje komunikaci mezi zařízením a vývojovým prostředím.
- *Nexus Client* – Jejich počet odpovídá počtu klientů (jader) přítomných v zařízení, obsahuje samotný ladicí nástroj.

Schéma celého systému je možné vidět na obrázku 4.4. Uvedené zařízení obsahuje dva procesory, které jsou přes *Nexus Client* připojeny k *Nexus Port*. Ten pak komunikuje s vývojovým prostředím prostřednictvím rozhraní JTAG.

[1] Toto FPGA je součástí výukové platformy FITKit [6]



Obrázek 4.4: Zapojení jednotlivých komponent ladicího nástroje a procesoru

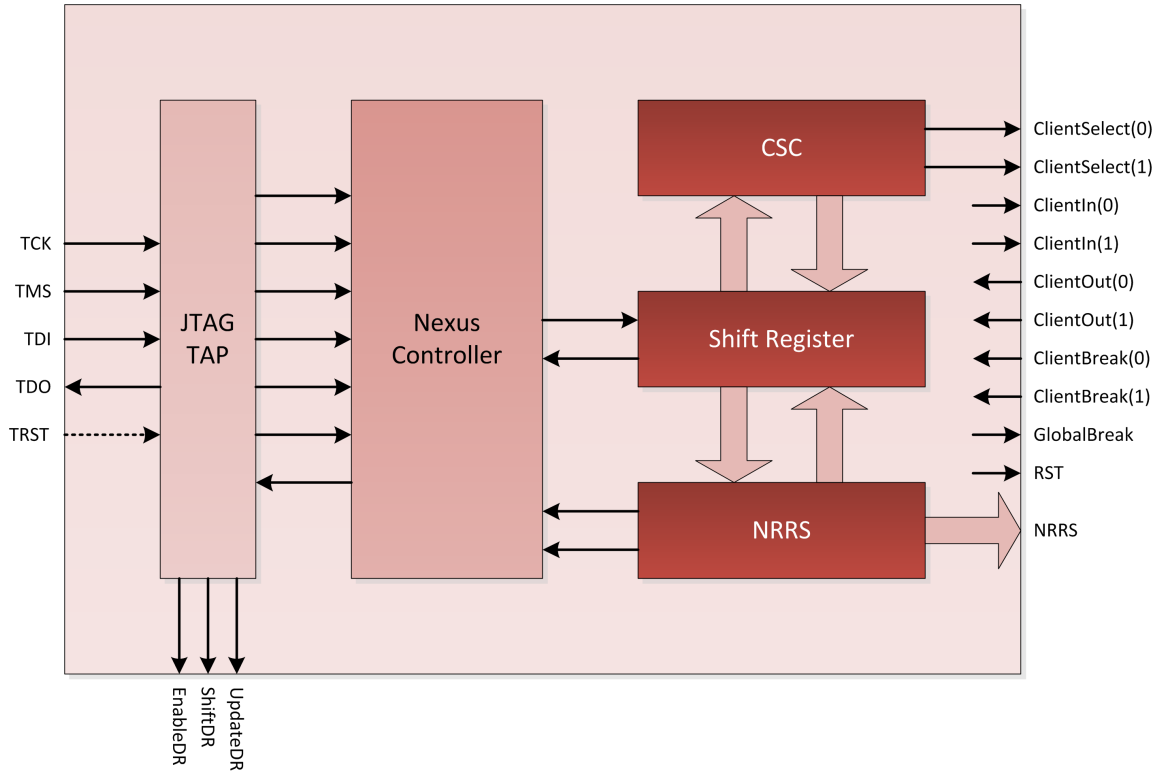
V následujících kapitolách bude nejprve detailně popsána implementace komponenty *Nexus Port* a poté i komponenty *Nexus Client*. Pro větší přehlednost zdrojových souborů jsem vytvořil balík `nexus`, který obsahuje všechny důležité konstanty sdílené mezi komponentami ladicího rozhraní. Ke každému registru obsaženém v obvodu obsahuje konstanty specifikující jeho velikost, význam jednotlivých bitů a implementované bity (viz přiložené zdrojové soubory).

4.2.1 Nexus Port

Nexus Port je rozhraní mezi jednotlivými klienty a vývojovým prostředím. Jeho vnitřní struktura je patrná z obrázku 4.5. Nejdůležitější částí je JTAG *Test Access Port*, který umožňuje komunikaci ladicího rozhraní a vývojového prostředí. Jeho implementace je mírně upravena přidáním instrukce NEXUS-ACCESS. Řídicí signály jsou napojeny na *Nexus Controller*, který má na starosti protokol čtení a zápisu do registrů (viz kapitola 3.4.1). *Nexus Port* dále obsahuje 8bitový posuvný registr a registry CSC (viz kapitola 3.4.2) a NRRS (viz obrázek 3.2).

Rozhraní

Entita `nexus_port` má poměrně bohaté rozhraní, které zahrnuje všechny signály JTAGu, některé řídicí signály z JTAG TAP, paralelní výstup registru NRRS a řídicí a datové signály vedoucí k jednotlivým klientům. Signál `ClientIn` sdružuje sériové datové vstupy ke každému klientovi, `ClientOut` naopak sériové výstupy z klientů. Řídicí signál `ClientSelect`



Obrázek 4.5: Vnitřní schéma komponenty *Nexus Port*

povoluje vždy jednoho klienta v závislosti na hodnotě v registru CSC. Signály `ClientBreak` jsou signalizace o aktivních breakpointech v jednotlivých klientech, `ClientERR` a `ClientDV` jsou pak stavové signály vedoucí k instrukčnímu registru TAP. Signál `GlobalBreak` je logickým součtem signálů `ClientBreak`.

```
entity nexus_port is
    generic (
        Manufacturer: std_logic_vector(10 downto 0) := "00000000001";
        PartNumber:   std_logic_vector(15 downto 0) := "0000000000000001";
        Version:      std_logic_vector(3 downto 0)  := "0001";
        ClientCnt:    natural                       := 1
    );
    port (
        -- JTAG Interface
        TDI:          in  std_logic;
        TMS:          in  std_logic;
        TCK:          in  std_logic;
        TRST:         in  std_logic;
        TDO:          out std_logic;
        -- JTAG Control Signals
        JTAGEnableDR: out std_logic;
        JTAGShiftDR:  out std_logic;
        JTAGUpdateDR: out std_logic;
        -- Nexus Data Signals
        NRRS:         out std_logic_vector(NRRS_SIZE-1 downto 0);
        -- Nexus Control Signals
        GlobalBreak:  out std_logic;
        RST:          out std_logic;
        -- Nexus Client Signals
        ClientIn:     out std_logic_vector(ClientCnt-1 downto 0);
        ClientOut:    in  std_logic_vector(ClientCnt-1 downto 0);
```

```
        ClientSelect: out std_logic_vector(ClientCnt-1 downto 0);
        ClientBreak:  in  std_logic_vector(ClientCnt-1 downto 0);
        ClientERR:    in  std_logic_vector(ClientCnt-1 downto 0);
        ClientDV:     in  std_logic_vector(ClientCnt-1 downto 0)
    );
end nexus_port;
```

Nexus Controller

Nexus Controller je řadič realizovaný jako konečný stavový automat. Rozhraní tvoří hodinový signál `CLK`, asynchronní reset `RST`, signál `NexusAccess` z instrukčního registru TAP signalizující načtenou instrukci NEXUS-ACCESS, kontrolní signály z TAP `UpdateIR` a `UpdateDR`. Z výstupních signálů jsou to potom stavový signál `RegSel`, který je aktivní, pokud se řadič nachází ve stavu *NRR_REG_SEL*, a signál `DataAcc` aktivní za stavu *NRR_DATA_ACC*.

```
entity nexus_ctrl is
    port (
        CLK:          in  std_logic;
        RST:          in  std_logic;
        NexusAccess:  in  std_logic;
        UpdateIR:     in  std_logic;
        UpdateDR:     in  std_logic;
        RegSel:       out std_logic;
        DataAcc:      out std_logic
    );
end nexus_ctrl;
```

Architektura této entity je popsána behaviorálně pomocí dvou procesů a dvou podmíněných přiřazení (podobně jako v případě řadiče TAP).

```
architecture nexus_ctrl_behav of nexus_ctrl is
    type state_t is (StIdle, StRegSel, StDataAcc);
    signal PresentState, NextState: state_t;
begin
    -- Present State Register
    present_state_reg: process (CLK)
    begin
        if (rising_edge(CLK)) then
            PresentState <= NextState;
        end if;
    end process;

    -- Next State Logic
    next_state_logic: process(PresentState, RST, NexusAccess, UpdateIR, ↩
        UpdateDR)
    begin
        case PresentState is
            when StIdle =>
                if (NexusAccess = '1') then
                    NextState <= StRegSel;
                else
                    NextState <= StIdle;
                end if;
            when StRegSel =>
                if (RST = '1') then
                    NextState <= StIdle;
                elsif (UpdateDR = '1') then
                    NextState <= StDataAcc;
                else
                    NextState <= StRegSel;
                end if;
            when StDataAcc =>
                if (RST = '1') then
```

```
                    NextState <= StIdle;
                elsif (UpdateDR = '1' or
                       (NexusAccess = '1' and UpdateIR = '1')) then
                    NextState <= StRegSel;
                else
                    NextState <= StDataAcc;
                end if;
            end case;
        end process;

        -- Output Logic
        RegSel  <= '1' when (PresentState = StRegSel) else '0';
        DataAcc <= '1' when (PresentState = StDataAcc) else '0';
    end nexus_ctrl_behav;
```

Posuvný registr

Důležitou roli nejen v případě komponenty *Nexus Port*, ale především *Nexus Client*, hraje posuvný registr. Jeho sériový vstup a výstup je napojen do TAP jako testovací řetězec. Posuvný registr `shift_reg` je vytvořen genericky, aby jej bylo možné použít s různou šířkou. Rozhraní tohoto registru obsahuje hodinový signál `CLK`, povolovací vstup `E`, sériový vstup `SI`, paralelní vstup `PI`, paralelní výstup `PO` a řídicí signál `Shift`.

```
    entity shift_reg is
        generic (
            Size: natural := 8
        );
        port (
            CLK:   in  std_logic;
            E:     in  std_logic;
            SI:    in  std_logic;
            PI:    in  std_logic_vector(Size-1 downto 0);
            PO:    out std_logic_vector(Size-1 downto 0);
            Shift: in  std_logic
        );
    end shift_reg;

    architecture shift_reg_behav of shift_reg is
        signal SR: std_logic_vector(Size-1 downto 0) := (others => '0');
    begin
        shift_reg: process (CLK)
        begin
            if (rising_edge(CLK)) then
                if (E = '1') then
                    if (Shift = '1') then
                        SR <= SI & SR(Size-1 downto 1);
                    else
                        SR <= PI;
                    end if;
                end if;
            end if;
        end process;

        PO <= SR;
    end shift_reg_behav;
```

Obsah registru je s náběžnou hranou hodin při aktivním povolovacím vstupu posouván o jeden bit směrem k nejméně významnému bitu.

Registr CSC

Pokud zařízení obsahuje více klientů, obsahuje *Nexus Port* také registr CSC, jehož paralelní vstup je připojen na posuvný registr. Podle hodnoty na výstupu tohoto registru jsou přepínány signály `ClientIn`, `ClientOut` a `ClientSelect`.

Registr NRRS

Součástí řídicí logiky realizující protokol přístupu k registrům uvnitř ladicího rozhraní je kromě řadiče *Nexus Controller* registr NRRS. V něm je uchovávána informace o čísle registru, do kterého je přistupováno, a jestli je z registru čteno, nebo do něj zapisováno. Stejně jako registr CSC je realizován entitou `data_reg`.

Vnitřní řídicí logika

Uvnitř entity `nexus_port` se nachází kromě výše uvedených komponent také řídicí logika. Ta se stará především o výběr klienta, pokud zařízení obsahuje více klientů.

```
client_gt1: if (ClientCnt > 1) generate
    client_select: for i in 0 to ClientCnt-1 generate
        ClientIn(i)     <= NexusIn when (conv_integer(CSC_Q) = i and
            DataAcc = '1' and NRRS_Q(NRRS_ADDR) /= NRR_CSC) else '0';
        ClientSelect(i) <= '1'     when (conv_integer(CSC_Q) = i and
            DataAcc = '1' and NRRS_Q(NRRS_ADDR) /= NRR_CSC) else '0';
    end generate;

    process (CSC_Q, DataAcc, NRRS_Q, ClientOut, SR_Q)
        variable Client: natural := 0;
    begin
        Client := conv_integer(CSC_Q);
        if (Client < ClientCnt and DataAcc = '1' and NRRS_Q(NRRS_ADDR) /=
            NRR_CSC) then
            NexusOut <= ClientOut(Client);
        else
            NexusOut <= SR_Q(0);
        end if;
    end process;
end generate;
client_eq1: if (ClientCnt = 1) generate
    ClientIn(0)     <= NexusIn      when (DataAcc = '1' and NRRS_Q(
        NRRS_ADDR) /= NRR_CSC) else '0';
    ClientSelect(0) <= '1'          when (DataAcc = '1' and NRRS_Q(
        NRRS_ADDR) /= NRR_CSC) else '0';
    NexusOut        <= ClientOut(0) when (DataAcc = '1' and NRRS_Q(
        NRRS_ADDR) /= NRR_CSC) else SR_Q(0);
end generate;
```

Kromě výběru klienta se zde nachází také multiplexor přepínající vstup do posuvného registru mezi registry CSC a NRRS a signály povolující zápis do těchto registrů.

```
SR_D <= CSC_Q when (NRRS_Q = NRR_CSC & NRR_R) else (others => '0');

NRRS_UE <= UpdateDR when (RegSel = '1') else '0';
CSC_UE  <= UpdateDR when (DataAcc = '1' and NRRS_Q = NRR_CSC & NRR_W) else
    '0';
```

Poslední důležitou funkcí je tvorba globální signalizace breakpointu a stavová signalizace do instrukčního registru.

```
GlobalBreak <= multi_or(ClientBreak);
```

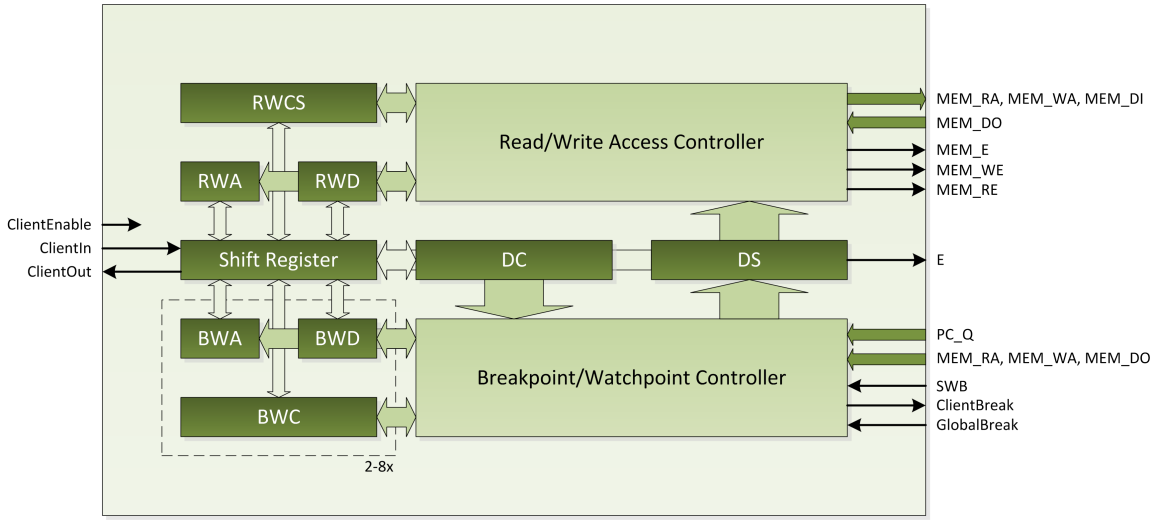
```
ERR <= multi_or(ClientERR);
DV  <= multi_or(ClientDV);
```

Pro snazší implementaci vícevstupého hradla OR byla navržena funkce `multi_or()`, která rekurzivně vytváří stromovou strukturu hradel OR.

```
function multi_or(slv: std_logic_vector) return std_logic is
    variable a: std_logic;
begin
    if (slv'length > 2) then
        a := multi_or(slv(slv'length-1 downto slv'length/2)) or
             multi_or(slv(slv'length/2-1 downto 0));
    elsif (slv'length = 2) then
        a := slv(slv'high) or slv(slv'low);
    else
        a := slv(slv'low);
    end if;

    return a;
end multi_or;
```

4.2.2 Nexus Client

Komponenta *Nexus Client* je o hodně složitější než *Nexus Port*. Obsahuje mnoho registrů – DC, DS, RWCS, RWA, RWD, BWC, BWA a BWD (význam viz kapitola 3.4). Všechny tyto registry jsou napojeny na posuvný registr SR. Kromě těchto registrů obsahuje entita `nexus_port` také řadič breakpointů *Breakpoint/Watchpoint Controller* a řadič přístupu do paměti *Read/Write Access Controller*. Vnitřní zapojení této entity ilustruje obrázek 4.6.



Obrázek 4.6: Vnitřní schéma komponenty *Nexus Client*

Rozhraní

Rozhraní komponenty *Nexus Client* ohsahuje řadu řídicích signálů z *Nexus Port* a také signály řídicí, stavové i datové z připojeného procesoru. Entita `nexus_client` je také řešena

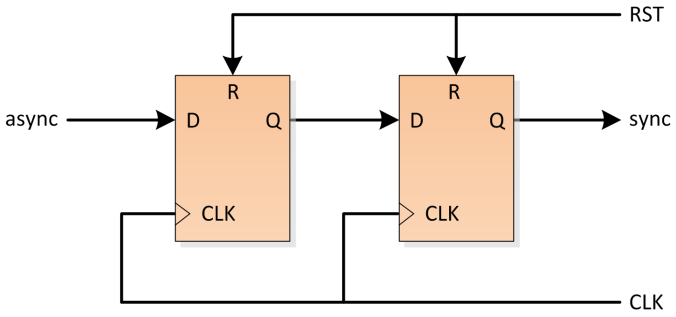
genericky, aby nebylo nutné při napojení na procesory s různou šířkou datové a adresové sběrnice entitu přepisovat.

Zajímavostí této entity je také to, že do ní vedou dva různé hodinové signály – jeden je společný s napojeným procesorem a druhý pochází z rozhraní JTAG. Tato skutečnost si vynucuje zvláštní způsob synchronizace, který bude dále v textu popsán.

Signály vedoucí k procesoru zahrnují především povolovací signál (slouží k zastavení činnosti procesoru), datový signál vedoucí z programového čítače procesoru a signály pro ovládání a sledování uživatelské paměti.

Synchronizace mezi dvěma hodinovými doménami

Protože v komponentě *Nexus Client* dochází k přenosu informace mezi dvěma hodinovými doménami, bylo nutné navrhnout mechanismus resynchronizace signálů, aby bylo zabráněno metastavům na výstupech registrů. Tato resynchronizace je založena na vložení dvou registrů do cesty signálu a tím snížení pravděpodobnosti metastavu [2]. Obrázek 4.7 takové zapojení ilustruje.

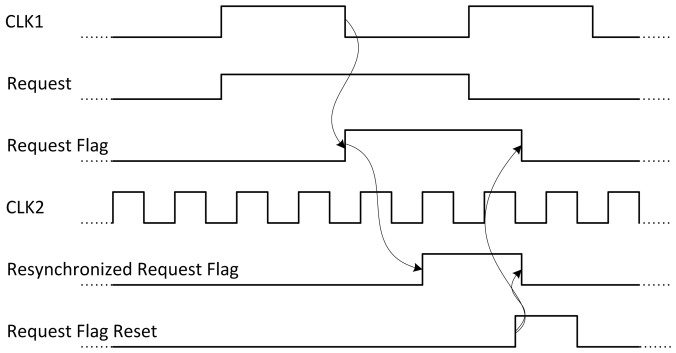


Obrázek 4.7: Resynchronizace signálu mezi hodinovými doménami

Pro větší znovupoužitelnost kódu jsem pro tento účel vytvořil novou entitu `resync`, jejíž rozhraní i vnitřní struktura přesně odpovídá obrázku 4.7.

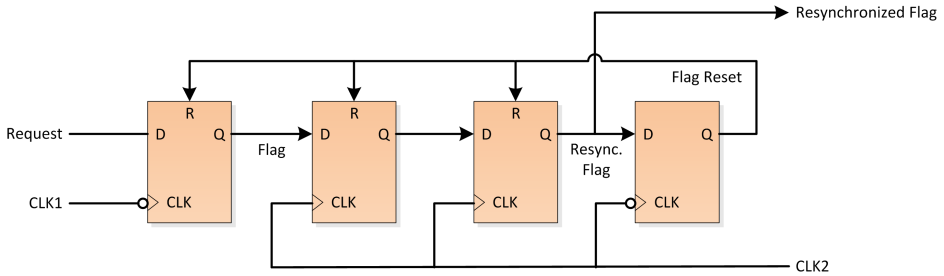
```
entity resync is
    port (
        CLK:   in  std_logic;
        RST:   in  std_logic;
        async: in  std_logic;
        sync:  out std_logic := '0'
    );
end resync;

architecture resync_behav of resync is
    signal tmp: std_logic := '0';
begin
    resync_regs: process (CLK)
    begin
        if (RST = '1') then
            tmp  <= '0';
            sync <= '0';
        elsif (rising_edge(CLK)) then
            tmp  <= async;
            sync <= tmp;
        end if;
    end process;
end resync_behav;
```

V několika případech je potřeba řešit další problém – při události v doméně s nižší frekvencí hodinového signálu je potřeba reagovat v doméně s vyšší frekvencí hodinového signálu pouze jednou. Za tímto účelem jsem navrhl jednoduchý protokol založený na principu *handshake*. Jeho časový diagram je možné vidět na obrázku 4.8.



Obrázek 4.8: Synchronizace žádosti generované v jiné hodinové doméně – časový diagram

Obvodové řešení (viz obrázek 4.9) pak zahrnuje již známý obvod pro resynchronizaci signálu a navíc ještě dva další registry – jeden pro generování žádosti, druhý pro reset příznaku žádosti.



Obrázek 4.9: Synchronizace žádosti generované v jiné hodinové doméně – obvodové řešení

Breakpoint/Watchpoint Controller

Jednotka *Breakpoint/Watchpoint Controller* realizovaná entitou `bw_ctrl` má na starosti vše, co se týká zastavování procesoru, ať už ručně nebo pomocí breakpointů, dále obsluhuje také watchpointy. Vstupem této jednotky jsou hodinové signály `TCK` a `CLK`, asynchronní reset `RST` a dále signály `Break` a `GlobalBreak` týkající se breakpointů, datové signály registrů DC, DS, BWC, BWA a BWD, řídicí a datové signály paměti procesoru, datový signál vedoucí z programového čítače procesoru `PC_Q` a signalizaci softwarového breakpointu `SWB`.

Až na obsluhu globálního a softwarového breakpointu pracuje celý obvod synchronně na sestupnou hranu hodinového signálu `CLK`. Část obvodu patřící do hodinové domény signálu `TCK` generuje žádosti o zastavení nebo obnovení činnosti procesoru pomocí mechanismu, který byl představen v předchozí kapitole.

```
-- Debug Stop/Exit Flags
process (TCK, DBS_R, DBE_R)
begin
    if (DBS_R = '1') then
        -- Reset Debug Stop Flag
        DBS_F <= '0';
    elsif (DBE_R = '1') then
        -- Reset Debug Exit Flag
        DBE_F <= '0';
    elsif (falling_edge(TCK)) then
        if (DC_UE = '1') then
            if (DC_Q(NRR_DC_DBE) = '0') then
                -- Debug mode disabled
                DBE_F <= '1';
            elsif (DC_Q(NRR_DC_DBR) = '0') then
                -- Request debug exit
                DBE_F <= '1';
            elsif (DC_Q(NRR_DC_DBE) = '1' and DC_Q(NRR_DC_DBR) = '1') then
                -- Request debug mode
                DBS_F <= '1';
            end if;
        end if;
     end if;
end process;
```

Hodinová doména signálu `CLK` obsahuje pouze jediný proces. Ten začíná asynchronní částí, tedy obsluhou globálního resetu, globálního a softwarového breakpointu. Při detekování breakpointu je do registru DS na pozici bitu DBS zapsána hodnota 1 a tím je procesor zastaven.

```
-- Breakpoints/Watchpoints
bw: process (RST, CLK, GlobalBreak, DS_Q, DC_Q, SWB)
    variable bw0: boolean;
    variable bw1: boolean;
begin
    if (RST = '1') then
        -- Global Reset
        for i in NRR_DS_BITS'range loop
            DS_Q(NRR_DS_BITS(i)) <= '0';
        end loop;
    elsif (DS_Q(NRR_DS_DBS) = '1' and DC_Q(NRR_DC_CBI) = '1' and
           DS_Q(NRR_DS_BP0) = '0' and DS_Q(NRR_DS_BP1) = '0' and
           GlobalBreak = '0') then
        -- Global Breakpoint down
        DS_Q(NRR_DS_DBS) <= '0';
    elsif (DC_Q(NRR_DC_CBI) = '1' and DC_Q(NRR_DC_DBE) = '1' and
           GlobalBreak = '1') then
        -- Global Breakpoint occured
        DS_Q(NRR_DS_DBS) <= '1';
    elsif (DC_Q(NRR_DC_DBE) = '1' and SWB = '1') then
        -- Software Breakpoint occured
        DS_Q(NRR_DS_DBS) <= '1';
        DS_Q(NRR_DS_SWB) <= '1';
```

Proces dále pokračuje obsluhou žádostí generovaných v hodinové doméně `TCK`. Zároveň je zde vyřešeno krokování, ke kterému dojde nejen při explicitním nastavením bitu SS v registru DC, ale také při každém opětovném spuštění činnosti procesoru po zastavení na breakpointu. Tím je schopen procesor překonat jeden hodinový cyklus, kdy je aktivní podmínka breakpointu a nezastavit se okamžitě.

```
    elsif (falling_edge(CLK)) then
        bw0 := false;
        bw1 := false;
```

```
                DBE_R <= '0';
                DBS_R <= '0';

                if (SS = '1') then
                    -- Single Step
                    SS <= '0';
                    if (DC_Q(NRR_DC_SS) = '1') then
                        DS_Q(NRR_DS_DBS) <= '1';
                        DS_Q(NRR_DS_SSS) <= '1';
                    end if;
                elsif (DBE_F_resync = '1') then
                    -- Debug Exit Flag
                    DS_Q(NRR_DS_DBS) <= '0';
                    DS_Q(NRR_DS_BP0) <= '0';
                    DS_Q(NRR_DS_BP1) <= '0';
                    DS_Q(NRR_DS_HWB) <= '0';
                    DS_Q(NRR_DS_SWB) <= '0';
                    DS_Q(NRR_DS_SSS) <= '0';
                    Break            <= '0';
                    SS               <= '1';
                    DBE_R            <= '1';
                elsif (DBS_F_resync = '1') then
                    -- Debug Stop Flag
                    DS_Q(NRR_DS_DBS) <= '1';
                    DBS_R            <= '1';
```

Zbytek procesu obsluhuje detekci breakpointů a watchpointů. Nejprve je obsloužen instrukční breakpoint.

```
                else
                    -- Breakpoint/Watchpoint 0
                    if (((BWC0_Q(NRR_BWC_BWE) = "01" and DC_Q(NRR_DC_DBE) = '1') or
                        BWC0_Q(NRR_BWC_BWE) = "11")) then
                        -- Breakpoint/Watchpoint 0 enabled
                        if (BWC0_Q(NRR_BWC_BWT) = '0') then
                            -- Compare for instruction types
                            if (BWC0_Q(NRR_BWC_BWO) = "10") then
                                -- Compare with BWA value
                                if (BWC0_Q(NRR_BWC_BRW) = "00" or
                                    BWC0_Q(NRR_BWC_BRW) = "10") then
                                    -- Break on read access
                                    if (PC_Q(AddressSize-1 downto 0) = BWA0_Q) then
                                        bw0 := true;
                                    end if;
                                end if;
                            end if;
                        else
```

Poté následuje detekce datových breakpointů – při operaci čtení a zápisu do paměti.

```
                            -- Compare for data types
                            if (BWC0_Q(NRR_BWC_BWO) = "10" or
                                BWC0_Q(NRR_BWC_BWO) = "11") then
                                -- Compare with BWA value
                                if (BWC0_Q(NRR_BWC_BRW) = "00" or
                                    BWC0_Q(NRR_BWC_BRW) = "10") then
                                    -- Break on read access
                                    if (MEM_WATCH_E = '1' and
                                        MEM_WATCH_RE = '1' and
                                        MEM_WATCH_RA = BWA0_Q) then
                                        bw0 := true;
                                    end if;
                                end if;
                                if (BWC0_Q(NRR_BWC_BRW) = "01" or
                                    BWC0_Q(NRR_BWC_BRW) = "10") then
                                    -- Break on write access
                                    if (MEM_WATCH_E = '1' and
```

```
                                    MEM_WATCH_WE = '1' and
                                    MEM_WATCH_WA = BWA0_Q) then
                                    bw0 := true;
                                end if;
                            end if;
                            if (BWC0_Q(NRR_BWC_BW0) = "01" or
                                BWC0_Q(NRR_BWC_BW0) = "11") then
                                -- Compare with BWD value
                                if (BWC0_Q(NRR_BWC_BRW) = "00" or
                                    BWC0_Q(NRR_BWC_BRW) = "10") then
                                    -- Break on read access
                                    if (MEM_WATCH_E = '1' and
                                        MEM_WATCH_DO = BWD0_Q) then
                                        bw0 := true;
                                    end if;
                                end if;
                                if (BWC0_Q(NRR_BWC_BRW) = "01" or
                                    BWC0_Q(NRR_BWC_BRW) = "10") then
                                    -- Break on write access
                                    if (MEM_WATCH_E = '1' and
                                        MEM_WATCH_DI = BWD0_Q) then
                                        bw0 := true;
                                    end if;
                                end if;
                            end if;
```

Jestli nastal breakpoint nebo watchpoint je rozlišeno až na konci procesu, protože watchpoint se od breakpointu liší pouze tím, že nezastavuje procesor.

```
                    if (bw0) then
                        if (BWC0_Q(NRR_BWC_BWE) = "01") then
                            -- Breakpoint occured
                            DS_Q(NRR_DS_BP0) <= '1';
                            DS_Q(NRR_DS_DBS) <= '1';
                            DS_Q(NRR_DS_HWB) <= '1';
                            Break            <= '1';
                        elsif (BWC0_Q(NRR_BWC_BWE) = "11") then
                            -- Watchpoint occured
                            DS_Q(NRR_DS_BP0) <= '1';
                        end if;
                    end if;
```

Předchozí blok podmínek je pak několikrát opakován podle počtu implementovaných breakpointů.

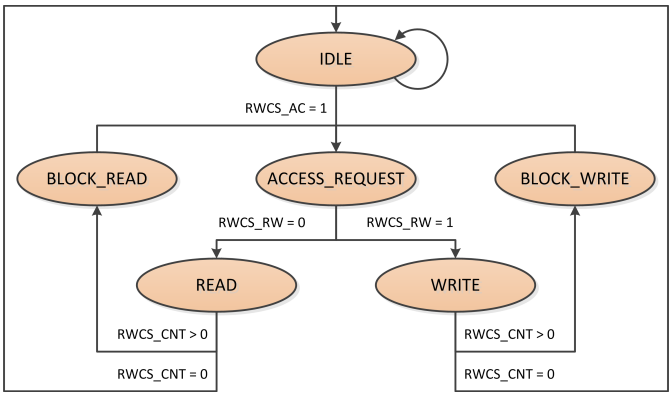
Read/Write Access Controller

Řadič přístupu do uživatelské paměti je řešen entitou `rwa_ctrl`, která je plně synchronní. Podobně jako `bw_ctrl` jsou zde ošetřeny 3 žádosti z hodinové domény `TCK`.

```
    -- RWCS, RWD Update/Read Flags
    process (TCK, RWCS_F, RWCS_R, RWD_UR, RWD_RR)
    begin
        if (RWCS_R = '1') then
            -- Reset RWCS Update Flag
            RWCS_F <= '0';
        elsif (RWD_UR = '1') then
            -- Reset RWD Update Flag
            RWD_UF <= '0';
        elsif (RWD_RR = '1') then
            -- Reset RWD Read Flag
            RWD_RF <= '0';
        elsif (falling_edge(TCK)) then
            if (RWCS_UE = '1') then
```

```
                -- Update RWCS
                RWCS_F <= '1';
            elsif (RWD_UE = '1') then
                -- Update RWD
                RWD_UF <= '1';
            elsif (RWD_REL = '1' and RWD_RE = '0') then
                -- Read RWD
                RWD_RF <= '1';
            end if;
            RWD_REL <= RWD_RE;
        end if;
    end process;
```

Samotný přístup do paměti je řízen stavovým automatem s 6 stavy (viz obrázek 4.10). Na počátku se řadič nachází v nečinném stavu *IDLE*. Po zapsání odpovídající hodnoty do registru RWCS je vygenerována žádost o přístup do paměti a automat změní stav na *ACCESS_REQUEST*. Poté vyhodnotí, jestli se jedná o požadavek na čtení, nebo zápis, nastaví příslušné řídicí signály k paměti a přesune se do stavu *READ* resp. *WRITE*. Nyní záleží na tom, jestli je požadováno blokové čtení resp. zápis. Pokud ano, řadič pokračuje stavem *BLOCK_READ* resp. *BLOCK_WRITE* a vrací se zpět do stavu *ACCESS_REQUEST*. Pokud se nejedná o blokový přístup nebo probíhá poslední cyklus přístupu, automat přejde přímo do stavu *IDLE*, ve kterém stabilně čeká na další požadavek.



Obrázek 4.10: Stavy a přechody konečného stavového automatu uvnitř `rwa_ctrl`

Implementace stavového automatu uvnitř tohoto řadiče se kvůli přehlednosti dosti liší od obvyklého přístupu, což ale ničemu nevadí, protože syntezátor korektně stavový automat rozpozná. Jediný proces v hodinové doméně `CLK` realizuje celý automat včetně výstupní logiky. Nejprve je ošetřen globální reset, při němž dojde k vynulování registru RWCS a nastavení výchozího stavu *IDLE*. Na sestupnou hranu hodinového signálu `CLK` pak reaguje zbytek obvodu. Řídicí signály k uživatelské paměti jsou v každém taktu vynulovány a pokud je aktivní některá z žádostí, je nastaven její resetovací signál.

```
    -- Read/Write Access
    rwa: process (CLK, RST)
    begin
        if (RST = '1') then
            for i in NRR_RWCS_BITS'range loop
                RWCS_Q(NRR_RWCS_BITS(i)) <= '0';
            end loop;
            State <= StIdle;
        elsif (falling_edge(CLK)) then
            MEM_E  <= '0';
```

```
MEM_RE <= '0';

...

if (RWCS_F_resync = '1') then
    RWCS_R <= '1';
end if;
if (RWD_RF_resync = '1') then
    RWD_RR <= '1';
end if;
if (RWD_UF_resync = '1') then
    RWD_UR <= '1';
end if;
```

Samotný proces přístupu do paměti probíhá pouze tehdy, je-li zastaven procesor. Jestliže je aktivní žádost o přístup, příštím stavem bude stav *ACCESS_REQUEST*.

```
if (DS_Q(NRR_DS_DBS) = '1') then
    -- Processor stopped
    if (RWCS_F = '1') then
        -- RWCS Update Enable
        for i in NRR_RWCS_BITS'range loop
            RWCS_Q(NRR_RWCS_BITS(i)) <= RWCS_D(NRR_RWCS_BITS(i));
        end loop;
        if (RWCS_D(NRR_RWCS_AC) = '1') then
            State <= StAccessRequest;
        else
            State <= StIdle;
        end if;
```

Ve stavu *ACCESS_REQUEST* začíná proces přístupu do paměti a tak je nutné nastavit řídicí signály k uživatelské paměti a rozhodnout, ve kterém stavu bude řadič pokračovat.

```
    elsif (State = StAccessRequest and
           RWCS_Q(NRR_RWCS_AC) = '1') then
        if (RWCS_Q(NRR_RWCS_RW) = '0') then
            -- Read Access
            MEM_E  <= '1';
            MEM_RE <= '1';
            RWD_WE <= '1';

            State <= StRead;
        elsif (RWCS_Q(NRR_RWCS_RW) = '1' and
               RWCS_Q(NRR_RWCS_DV) = '1') then
            -- Write Access
            MEM_E  <= '1';
            MEM_WE <= '1';

            State <= StWrite;
        end if;
```

Přístup do paměti může být buď blokový (pokud je hodnota CNT v registru RWCS větší jak 0) nebo můžeme požadovat pouze jednu hodnotu. V takovém případě je po uložení hodnoty proces ukončen a řadič navrácen do stavu *IDLE*.

```
    elsif (State = StRead) then
        -- Read Access
        if (RWCS_Q(NRR_RWCS_CNT) = (RWCS_Q(NRR_RWCS_CNT)'high ↩
            downto RWCS_Q(NRR_RWCS_CNT)'low => '0')) then
            RWCS_Q(NRR_RWCS_AC)  <= '0';
            State                <= StIdle;
        else
            RWCS_Q(NRR_RWCS_CNT) <= RWCS_Q(NRR_RWCS_CNT)-1;
            RWA_WE               <= '1';
            State                <= StBlockRead;
```

```
                    end if;

                    RWCS_Q(NRR_RWCS_DV)  <= '1';
                    RWCS_Q(NRR_RWCS_ERR) <= '0';
                elsif (State = StWrite) then
                    -- Write Access
                    if (RWCS_Q(NRR_RWCS_CNT) = (RWCS_Q(NRR_RWCS_CNT)'high ↩
                        downto RWCS_Q(NRR_RWCS_CNT)'low => '0')) then
                        RWCS_Q(NRR_RWCS_AC)  <= '0';
                        State                <= StIdle;
                    else
                        RWCS_Q(NRR_RWCS_CNT) <= RWCS_Q(NRR_RWCS_CNT)-1;
                        RWA_WE               <= '1';
                        State                <= StBlockWrite;
                    end if;

                    RWCS_Q(NRR_RWCS_DV)  <= '0';
                    RWCS_Q(NRR_RWCS_ERR) <= '0';
```

Pokud požadujeme blokový přístup, adresa v registru RWA je postupně inkrementována, hodnota CNT v registru RWCS naopak dekrementována a při každém zapsání resp. přečtení registru RWD je vykonán jeden cyklus přístupu.

```
                elsif (State = StBlockRead) then
                    -- Block Read Access in progress
                    if (RWD_RF = '1') then
                        State  <= StAccessRequest;
                    end if;
                elsif (State = StBlockWrite) then
                    -- Block Write Access in progress
                    if (RWD_UF = '1') then
                        RWCS_Q(NRR_RWCS_DV) <= '1';
                        State               <= StAccessRequest;
                    end if;
                end if;
```

Vnitřní řídicí logika

Zbytek komponenty *Nexus Client* již tvoří pouze kontrolní signály a multiplexory k registrům.

```
    -- Update Enable Signals
    WE      <= '1' when (ClientEnable = '1' and UpdateDR = '1' and
                         NRRS(NRRS_RW) = NRR_W) else '0';

    DC_UE   <= '1' when (WE = '1' and NRRS(NRRS_ADDR) = NRR_DC)   else '0';
    RWCS_UE <= '1' when (WE = '1' and NRRS(NRRS_ADDR) = NRR_RWCS) else '0';

    ...

    -- Shift Register
    ClientOut <= SR_Q(SR_SIZE-NRR_DC_SIZE) when (NRRS(NRRS_ADDR) = NRR_DC) else
                 SR_Q(SR_SIZE-NRR_DS_SIZE) when (NRRS(NRRS_ADDR) = NRR_DS) else

                 ...

    RE     <= '1' when (ClientEnable = '1' and NRRS(NRRS_RW) = NRR_R) else '0';

    RWD_RE <= '1' when (RE = '1' and NRRS(NRRS_ADDR) = NRR_RWD) else '0';

    SR_D   <= DC_Q & zeros(SR_SIZE-NRR_DC_SIZE) when
                  (RE = '1' and NRRS(NRRS_ADDR) = NRR_DC) else
              DS_Q & zeros(SR_SIZE-NRR_DS_SIZE) when
                  (RE = '1' and NRRS(NRRS_ADDR) = NRR_DS) else
```

```
                RWCS_Q & zeros(SR_SIZE-NRR_RWCS_SIZE) when
                    (RE = '1' and NRRS(NRRS_ADDR) = NRR_RWCS) else

                ...
```

Výstupy všech registrů obsažených v entitě `nexus_client` jsou napojeny na posuvný registr SR. Registry však nemají stejnou šířku a tak musejí být některé datové signály doplněny zprava nulami pomocí funkce `zeros()`.

```
function zeros(size: natural) return std_logic_vector is
    variable res: std_logic_vector(size-1 downto 0);
begin
    res := (others => '0');
    return res;
end zeros;
```

4.3 Napojení ladicího rozhraní na generovaný procesor

Vzhledem k univerzálnosti ladicího rozhraní je napojení na procesor poměrně snadné. Kromě napojení datových signálů k registrům procesoru a uživatelské paměti je potřeba pouze ošetřit zastavování procesoru. V případě procesoru generovaného nástroji projektu Lissom bylo nutné rozvést signál `E` (*Enable*) do řadiče procesoru a všech funkčních jednotek. Díky tomu, že řadič procesoru i všechny funkční jednoty pracují na principu konečného stavového automatu, stačilo signál `E` zavést do každého z procesů `proc_state`.

```
proc_state: process (CLK, RST)
begin
    if (RST = '0') then
        main_state_reg <= "001";
        main_pipeline_EX_state_reg <= "001";
        main_pipeline_FE_state_reg <= "0001";
        main_pipeline_ID_state_reg <= "001";
        main_pipeline_ST_state_reg <= "01";
    elsif (CLK'EVENT and CLK = '1') then
        if (E = '1') then
            main_state_reg <= main_next_state_reg;
            main_pipeline_EX_state_reg <= main_pipeline_EX_next_state_reg;
            main_pipeline_FE_state_reg <= main_pipeline_FE_next_state_reg;
            main_pipeline_ID_state_reg <= main_pipeline_ID_next_state_reg;
            main_pipeline_ST_state_reg <= main_pipeline_ST_next_state_reg;
        end if;
    end if;
end process proc_state;
```

Další úpravou byla změna řídicích signálů paměti a multiplexorů datových signálů do paměti. Pokud je signál `E` v nule, procesor neběží a přístup do paměti je umožněn ladicímu nástroji.

```
data_mem_Enable <=
    (not E and data_mem_Enable_main) or
    (E and (data_mem_Enable_ex_output_memory_ex_output_memory_1_behavior_1 ↩
        or
    data_mem_Enable_id_memory_access_id_memory_access_1_behavior_1));

...

with sig_32 select
data_mem_Data_in <=
    data_mem_Data_in_ex_output_memory_ex_output_memory_1_behavior_1 when "↩
        100",
```

```
            data_mem_Data_in_id_memory_access_id_memory_access_1_behavior_1 when "↩
                010",
            data_mem_Data_in_main when "001" | "101" | "011" | "111",
            (others => '0') when others;
```

Signály vedoucí z multiplexorů přímo do paměti jsou pak vyvedeny přes rozhraní procesoru do ladicího nástroje, aby mohly být sledovány pomocí jednotky *Breakpoint/Watchpoint Controller*.

4.4 Syntéza

Tabulka 4.2 ukazuje obsazenost logiky po syntéze samotného ladicího rozhraní v programu Xilinx ISE. Stejně jako u syntézy rozhraní JTAG je zde srovnána obsazenost FPGA Xilinx Spartan XC3S50-4PQ208C při optimalizaci na rychlost a na plochu. Celé ladicí rozhraní tedy zabírá cca 40 % plochy tohoto FPGA.

Typ logiky	**Optimalizace na rychlost**	**Optimalizace na plochu**
Slice Flip Flops	232	228
4vstupové LUT	499	465
Obsazené Slice	328	310

Tabulka 4.2: Obsazenost logiky po syntéze ladicího rozhraní v programu Xilinx ISE

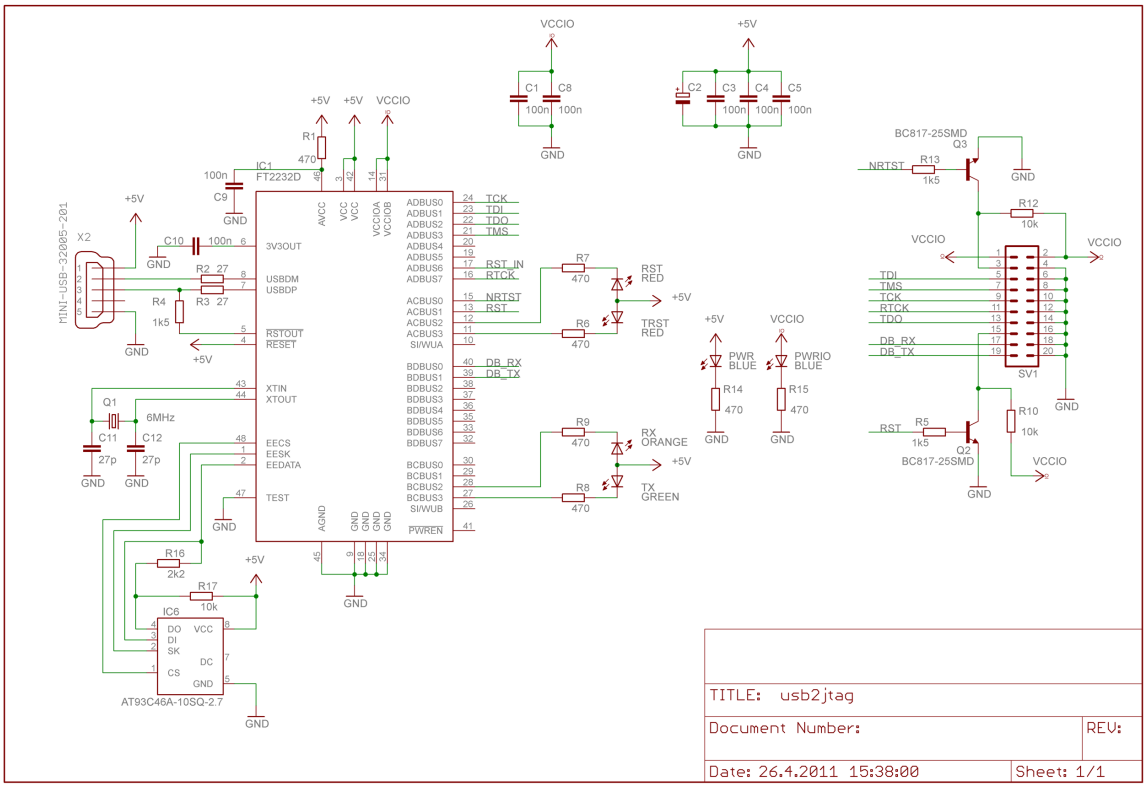
Kapitola 5

Konstrukce JTAG adaptéru

Aby bylo možné s ladicím rozhraním procesoru komunikovat, je nutné mít k dispozici JTAG adaptér. Ten slouží jako prostředník mezi počítačem s vývojovým prostředím a procesorem.

Konstrukcí JTAG adaptéru existuje celá řada, je možné zakoupit mnoho komerčních řešení – mezi nimi např. Altera USB-Blaster a Xilinx Platform Cable, oba komunikující přes sběrnici USB. Existují však i adaptéry založené na paralelním portu.

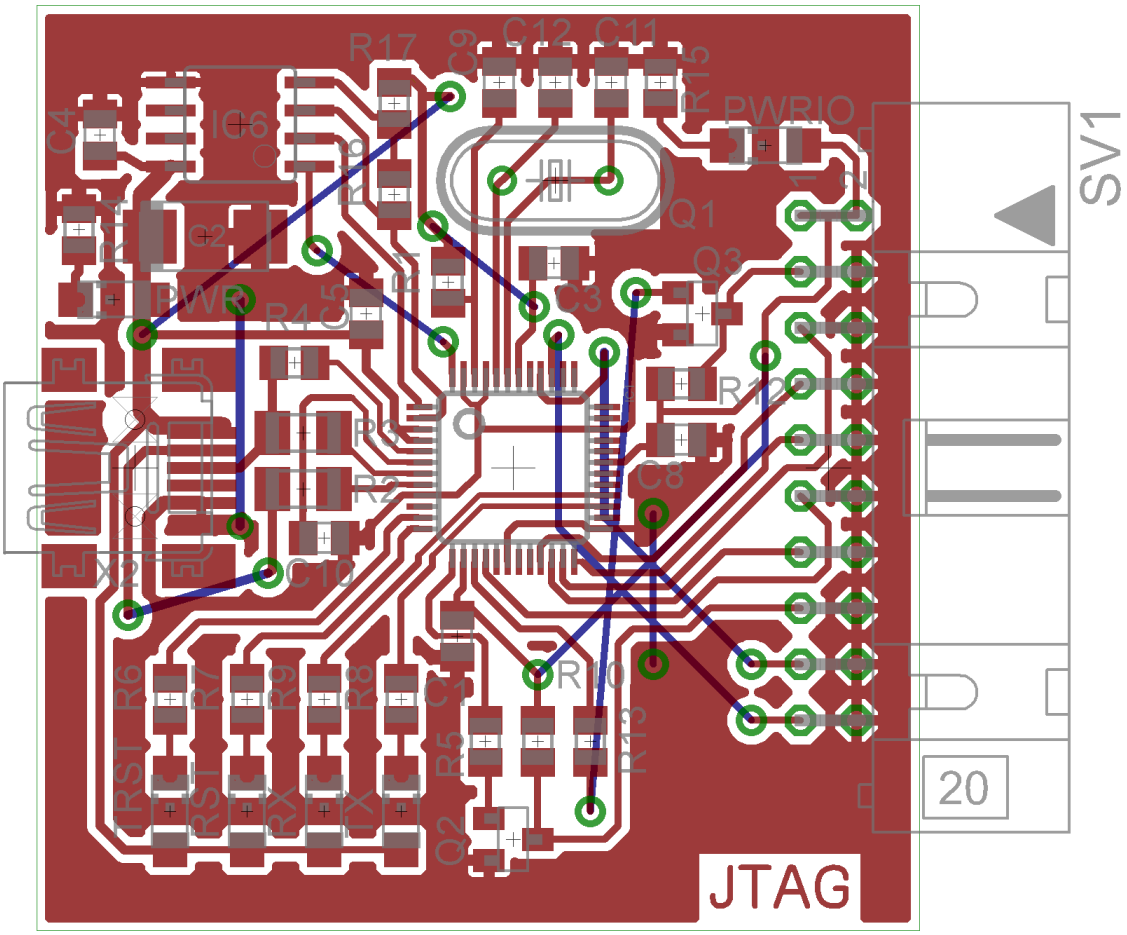
Má konstrukce adaptéru využívá obvod FT2232 firmy Future Technology Devices International Ltd., který funguje jako převodník sběrnice USB na sériový port, ale dokáže fungovat i ve speciálních režimech podporujících komunikační protokoly jako např. SPI, I2C a JTAG. Schéma zapojení je možné vidět na obrázku 5.1.



Obrázek 5.1: Schéma JTAG adaptéru

Kanál 1 obvodu FT2232 je vyveden na konektor SV1, obsahuje signály rozhraní JTAG – `TCK`, `TMS`, `TDI` a `TDO`. Kromě těchto signálů je zde vyveden ještě signál `RTCK` (*Return Clock*), v některých zapojeních také používaný. Druhý kanál pak funguje jako obyčejný sériový port. K obvodu FT2232 je dále připojena paměť EEPROM, která může uchovávat jednoznačnou identifikaci zařízení.

Návrh desky plošných spojů byl proveden s ohledem na velikost celého zařízení a jednoduchost výroby v domácích podmínkách. Proto je deska plošných spojů navržena jako jednostranná. Celý návrh byl proveden pomocí programu Eagle firmy CadSoft Computer GmbH.



Obrázek 5.2: Návrh desky plošných spojů JTAG adaptéru

Kapitola 6

Testování

Souběžně s návrhem ladicího rozhraní bylo nutné průběžně navržený obvod testovat. To lze provést dvěma způsoby – simulací obvodu v běžně dostupném simulátoru (např. ISim od firmy Xilinx nebo Modelsim od firmy Mentor Graphics) nebo přímo na FPGA.

6.1 Simulace

6.1.1 JTAG

Vzhledem k postupnému návrhu od rozhraní JTAG přes jednotlivé komponenty ladicího nástroje jsem nejprve vytvořil testovací obvod (tzv. *testbench*) pro JTAG TAP.

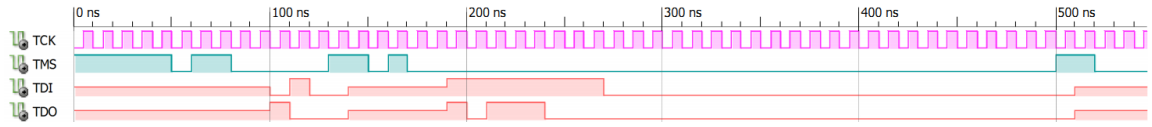
```
architecture tap_tb_behav of tap_tb is

    ...
begin
    tap_instance: entity work.tap
        generic map (
            DataSize     => 16,
            Manufacturer => "00000001110",
            PartNumber   => "0000000000000000",
            Version      => "0000"
        )
        port map (
            TDI => TDI,
            TMS => TMS,
            TCK => TCK,

            ...
        );

    TCK <= not TCK after TCK_PERIOD/2;

    -- Test Process
    testbench: process
    begin

        ...

    end process;
end tap_tb_behav;
```

Abych zpřehlednil a zjednodušil návrh testů, doplnil jsem balíček `jtag` o několik procedur zapouzdřujících elementární funkce rozhraní JTAG. Procedura `jtag_reset()` provede reset rozhraní zapsáním pěti jedniček na signál `TMS`.

```
        procedure jtag_reset (
            signal TDI: out std_logic;
            signal TMS: out std_logic;
            signal TCK: in  std_logic
        ) is
        begin
            TDI <= '0';
            TMS <= '1';
            wait until falling_edge(TCK);
            wait until falling_edge(TCK);
            wait until falling_edge(TCK);
            wait until falling_edge(TCK);
            wait until falling_edge(TCK);
            TMS <= '0';
            wait until falling_edge(TCK);
        end jtag_reset;
```

K provedení instrukčního scanu slouží procedura `jtag_irscan()`, datový scan pak provede procedura `jtag_drscan()`. Tyto procedury je možné nalézt v přiložených zdrojových kódech. Za pomoci těchto procedur je pak možné efektivně testovat navržený obvod – příklad takového testu je uveden v následujícím výpisu a výsledek simulace na obrázku 6.1.

```
        testbench: process
            variable idcode: std_logic_vector(31 downto 0);
        begin
            jtag_reset(TDI, TMS, TCK);
            jtag_irscan(TDI, TMS, TCK, I_IDCODE);
            jtag_drscan(TDI, TMS, TCK, TDO, "00000000000000000000000011111111", ↩
                idcode, 32);
            wait;
        end process;
```



Obrázek 6.1: Test instrukce IDCODE

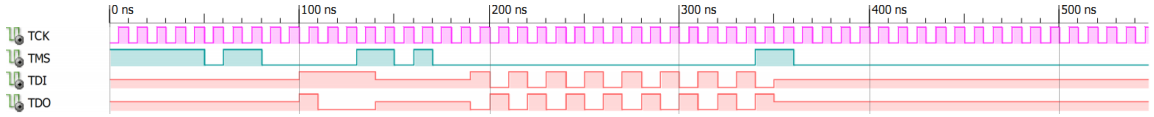
Ve výsledku simulace je jasně vidět vysouvání obsahu identifikačního registru. Dalším testem je možno ověřit funkčnost testovacího řetězce (obrázek 6.2).

```
        testbench: process
            variable data: std_logic_vector(15 downto 0);
        begin
            jtag_irscan(TDI, TMS, TCK, I_EXTEST);
            jtag_drscan(TDI, TMS, TCK, TDO, "0101010101010101", data, 16);
            wait;
        end process;
```

Obrázek 6.2: Načtení testovacího řetězce

Poslední test ověřuje vlastnosti instrukce BYPASS. Po nasunutí instrukce BYPASS je proveden datový scan, z časového průběhu signálu `TDO` je dobře vidět, že je nejprve hodnota na výstupu 0 a poté je ven vysouván vstupní signál `TDI` (viz obrázek 6.3). Naopak z obrázku 6.1 je patrné, že identifikační řetězec začíná hodnotou 1. Tím je umožněno automatické rozpoznání zařízení v řetězci.

```
testbench: process
    variable data: std_logic_vector(15 downto 0);
begin
    jtag_irscan(TDI, TMS, TCK, I_BYPASS);
    jtag_drscan(TDI, TMS, TCK, TDO, "0101010101010101", data, 16);
    wait;
end process;
```



Obrázek 6.3: Test instrukce BYPASS

6.1.2 Nexus 5001

Podobně jako u balíčku `jtag` jsem usnadnil tvorbu testbenchů ladicího rozhraní napsáním několika elementárních procedur v balíčku `nexus`. Procedura `nexus_nrr_read()` přečte obsah registru `NRR` a uloží jej do proměnné `NRR_OUT`.

```
procedure nexus_nrr_read (
    signal   TDI:     out std_logic;
    signal   TMS:     out std_logic;
    signal   TCK:     in  std_logic;
    signal   TDO:     in  std_logic;
    constant NRR:     in  std_logic_vector;
    variable NRR_OUT: out std_logic_vector
) is
    variable NRR_IN: std_logic_vector(NRR_OUT'length-1 downto 0) := (others↩
        => '0');
begin
    jtag_drscan(TDI, TMS, TCK, TDO, conv_range(NRR & NRR_R));
    jtag_drscan(TDI, TMS, TCK, TDO, NRR_IN, NRR_OUT);
end nexus_nrr_read;
```

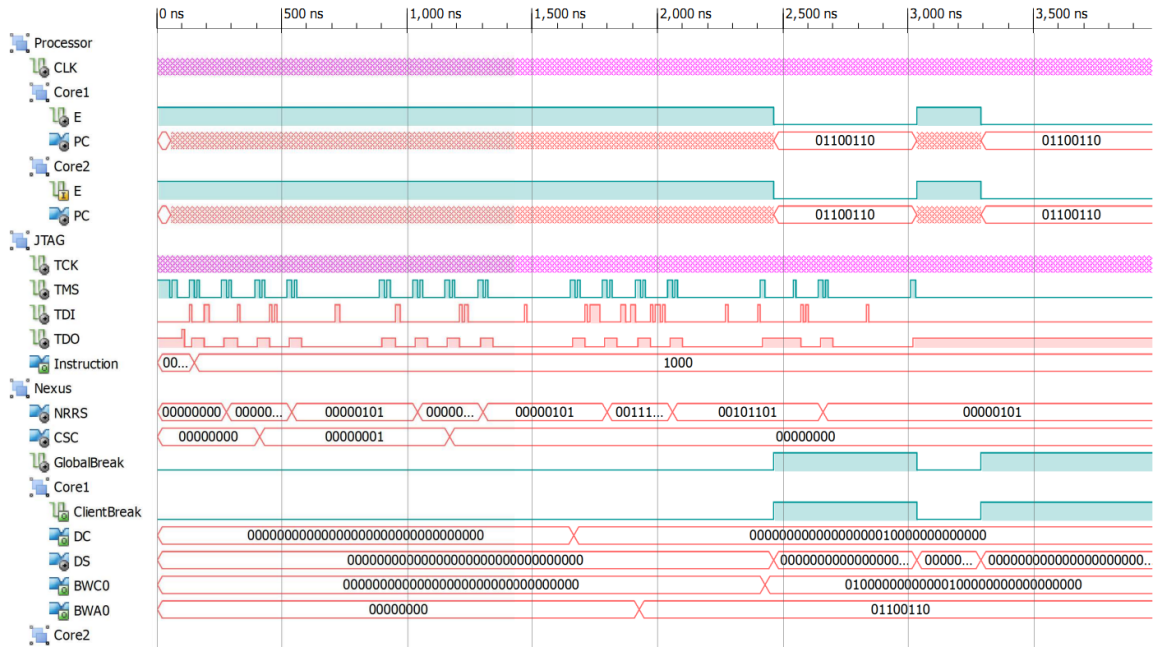
Podobně druhá procedura `nexus_nrr_write()` do registru `NRR` uloží hodnotu `NRR_IN`.

```
procedure nexus_nrr_write (
    signal   TDI:    out std_logic;
    signal   TMS:    out std_logic;
    signal   TCK:    in  std_logic;
    signal   TDO:    in  std_logic;
    constant NRR:    in  std_logic_vector;
    constant NRR_IN: in  std_logic_vector
) is
begin
    jtag_drscan(TDI, TMS, TCK, TDO, conv_range(NRR & NRR_W));
    jtag_drscan(TDI, TMS, TCK, TDO, conv_range(NRR_IN));
end nexus_nrr_write;
```

Oproti testům rozhraní JTAG jsou testy ladicího nástroje o dost náročnější. Součástí testovacího obvodu jsou dvě imitace procesoru (pouze programový čítač a paměť), dvě komponenty *Nexus Client* a jeden *Nexus Port*. Ze všech provedených testů, které jsou součástí přílohy, zde uvedu pouze několik.

Prvním z nich je ověření funkčnosti instrukčního breakpointu. Zároveň je testován globální breakpoint, který je povolen bitem CBI v registru DC na klientu 1. Instrukční breakpoint na klientu 0 je pak nastaven pomocí registrů DC, BWA0 a BWC0. Ze simulace je vidět zastavení obou procesorů na adrese 01100110 v programovém čítači. Po uplynutí 10 period hodin `TCK` je pak procesor ručně opět uveden do činnosti a po cca 300 ns opět dojde k instrukčnímu breakpointu. Obrázek 6.4 zahrnuje i časové průběhy signálů JTAG.

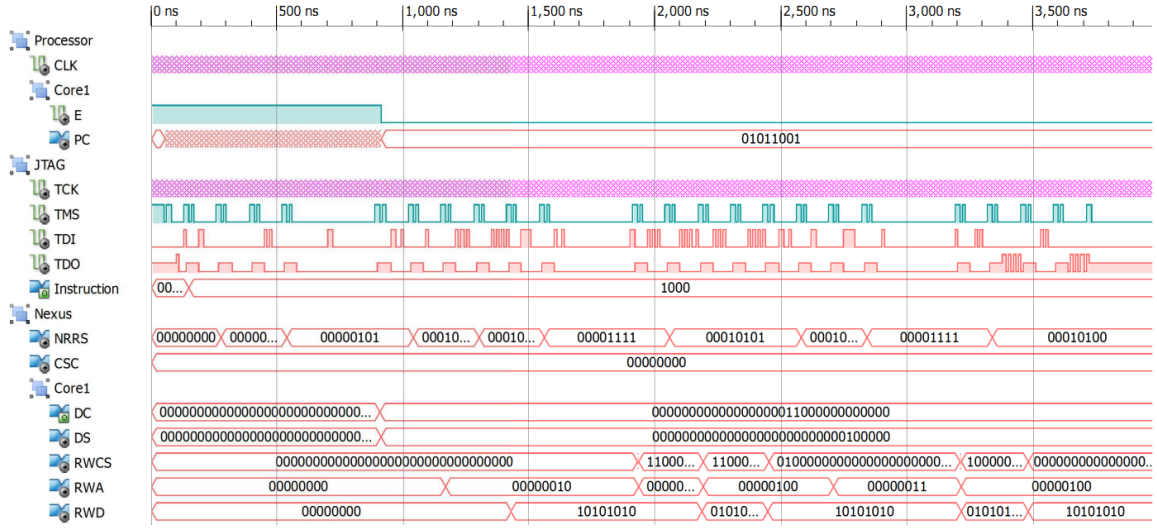
```
    testbench: process
        variable data: std_logic_vector(7 downto 0);
        variable test: natural;
    begin
        jtag_reset(TDI, TMS, TCK);
        jtag_irscan(TDI, TMS, TCK, TDO, I_NEXUS_ACCESS);

        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_CSC, "00000001");
        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_DC, "↩
            00000000000000000110000000000000");

        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_CSC, "00000000");
        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_DC, "↩
            00000000000000000010000000000000");
        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_BWA0, "01100110");
        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_BWC0, "↩
            01000000000001000000000000000000");
        wait for 10*TCK_PERIOD;
        nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_DC, "↩
            00000000000000000010000000000000");
    end process;
```



Obrázek 6.4: Test instrukčního breakpointu

Druhý test, který zde uvedu, se týká jednotky *Read/Write Access Controller* a pracuje pouze s klientem 0. Procesor je nejprve ručně zastaven zapsáním bitu DBR do registru DC a poté je zahájen blokový zápis do paměti (viz kapitola 3.4.5). Nejprve je do registru RWA zapsána počáteční adresa a do registru RWD první blok dat. Poté je přenos inicializován nastavením registru RWCS a dalšími zápisy do registru RWD je přenos uskutečněn. Následně je provedeno blokové čtení.

```
testbench: process
    variable data: std_logic_vector(7 downto 0);
    variable test: natural;
begin
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_CSC, "00000000");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_DC, "↩
        00000000000000000011000000000000");

    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWA, "00000010");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWD, "10101010");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWCS, "↩
        11000000000000000000000000001001");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWD, "01010101");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWD, "10101010");

    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWA, "00000011");
    nexus_nrr_write(TDI, TMS, TCK, TDO, NRR_RWCS, "↩
        10000000000000000000000000000100");
    nexus_nrr_read(TDI, TMS, TCK, TDO, NRR_RWD, 8);
    nexus_nrr_read(TDI, TMS, TCK, TDO, NRR_RWD, 8);
end process;
```

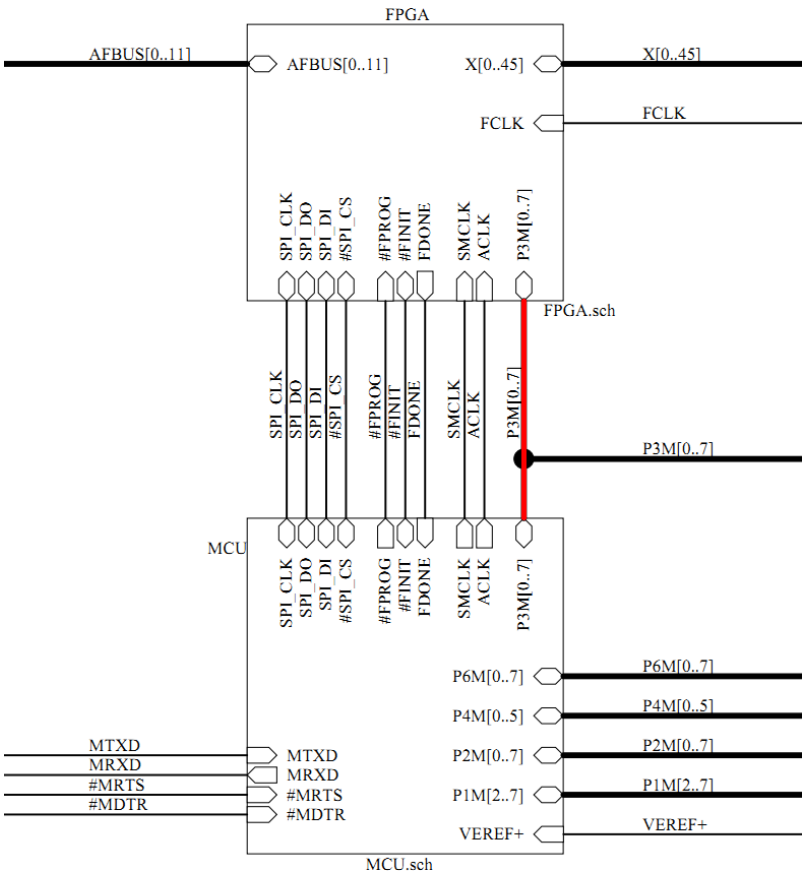


Obrázek 6.5: Test přístupu do paměti

6.2 Platforma FITKit

FITKit [6] je vývojový kit obsahující mikrokontrolér MSP430 firmy Texas Instruments, FPGA XC3S50-4PQ208C řady Spartan 3 firmy Xilinx a řadu periférií. Díky kombinaci programovatelného hradlového pole a mikrokontroléru se velmi hodí na testování nejrůznějších návrhů vestavěných systémů.

Kvůli zjednodušení testování rozhraní JTAG a ladicího nástroje jsem využil propojení mikrokontroléru s FPGA sběrnicí P3M (viz obrázek 6.6). Přes tuto sběrnici jsem propojil signály `TCK`, `TMS`, `TDI` a `TDO` rozhraní JTAG. Na mikrokontroléru MSP430 jsem pak napsal elementární funkce pro komunikaci přes rozhraní JTAG (viz příloha). Tyto příkazy jsem poté spouštěl přes skriptovatelný terminál QDevKit [6].



Obrázek 6.6: Propojení FPGA a MCU na platformě FITKit [6]

6.2.1 JTAG

Nejprve jsem se zaměřil na testování rozhraní JTAG. Testovací řetězec o délce jednoho bitu jsem nejprve napojil na LED diodu D4. Pomocí datového scanu jsem byl schopný tuto LED diodu ovládat. Po ověření této základní funkčnosti jsem testovací řetězec rozšířil na 8 bitů a přes dekodér pro segmentový displej jsem jej vyvedl na sběrnici X. Na nepájivé kontaktní pole jsem umístil dva segmentové displeje a ty jsem připojil na sběrnici X. Příkazy zadávanými ve skriptovatelném terminálu jsem ovládal hodnotu na segmentových displejích.

Správné časování signálů rozhraní JTAG jsem následně ověřil připojením logického analyzátoru na sběrnici P3M.

6.2.2 Nexus 5001

Ladicí rozhraní jsem nejprve otestoval napojením na imitaci procesoru (viz simulace). Programový čítač jsem vyvedl opět na sběrnici X a zobrazil pomocí segmentových displejů. Po odladění tohoto jednoduchého systému jsem přistoupil k napojení ladicího rozhraní na generovaný procesor Codea. Na tomto procesoru byl implementován filtr FIR 4. řádu typu dolní propust. Vstup tohoto filtru jsem napojil přes rozhraní SPI na A/D sigma delta převodník a výstup pak na D/A převodník kodeku TLV320AIC23B, který se nachází na platformě FITKit a analogový vstup a výstup je vyveden na audio konektory na desce. Ladicí rozhraní se bohužel na FPGA spolu s procesorem Codea nevešlo celé a tak jsem musel odstranit jednotku *Read/Write Access Controller* a jeden breakpoint. Poté se mi podařilo procesor spolu s ladicím rozhraním nahrát do FPGA a otestovat jeho funkčnost.

Pro testování celého ladicího rozhraní na víceprocesorovém systému jsem si zapůjčil rozšířenou verzi FITKitu s FPGA XC3S400, který obsahuje 8x více logických hradel (tj. 400 000). Systém obsahoval dva procesory Codea, každý zpracovával jeden kanál zvuku, dvě komponenty *Nexus Client* a jeden *Nexus Port*. Příkazy v skriptovatelném terminálu jsem pak ovládal oba procesory a otestoval jsem veškerou funkčnost nabízenou ladicím rozhraním, tedy i globální breakpoint, blokový přístup do paměti a jiné funkce.

Kapitola 7

Závěr

Cílem této práce bylo navrhnout a realizovat univerzální ladicí rozhraní procesoru a jeho napojení na procesor generovaný nástroji projektu Lissom. Zadání se mi podařilo splnit až na automatické generování tohoto napojení, to nebylo možné vzhledem k probíhajícím úpravám kódu generátoru hardware. Nad rámec zadání jsem navrhl a otestoval JTAG adaptér.

Navržený nástroj umožňuje plnohodnotně ladit aplikace na vestavěných zařízeních popsaných v jazyce ISAC. Podporuje běžné prostředky pro řízení a sledování činnosti procesoru, přístupu k uživatelské paměti a registrům. Všechny tyto funkce jsou přístupné pomocí rozhraní JTAG, které navíc může být využito i k jiným účelům (např. testování hardware).

Souběžně s vývojem ladicího nástroje na čipu probíhal také vývoj simulátoru. Prostředky nabízené ladicím rozhraním simulátoru jsou velmi podobné těm na čipu, vycházejí také ze standardu Nexus 5001. Stejně jako mnou navržený nástroj podporuje i simulátor všechny funkce první třídy standardu Nexus 5001, nabízí však i některé funkce navíc (především větší možnosti nastavení breakpointů).

Implementací ladicího rozhraní má práce na projektu Lissom zdaleka nekončí. Aby bylo možné navržené rozhraní plně využívat, bude jej nutné propojit s vývojovým prostředím. Nadále budu sledovat vývoj ladicího rozhraní simulátoru a přizpůsobovat rozhraní na čipu tak, aby byla zajištěna co nejvyšší míra ekvivalence mezi simulovaným modelem procesoru a generovaným hardware.

Tato práce pro mě měla nesmírný přínos, neboť jsem si vyzkoušel návrh, implementaci a testování poměrně složitého číslicového obvodu. Díky projektu Lissom jsem měl navíc možnost pracovat v početném týmu. Při realizaci jsem narazil na mnoho komplikací, které se mi podařilo zvládnout a získal jsem tak mnoho cenných zkušeností v oblasti návrhu číslicových obvodů.

Literatura

[1] *The Nexus 5001 Forum™*. [b.m.]: IEEE- Industry Standards and Technology Organization (IEEE-ISTO), 2003.

[2] Šťastný, J. *FPGA prakticky – Realizace číslicových systémů pro programovatelná hradlová pole*. 1. vydání. Praha: BEN - technická literatura, 2010. ISBN 978-80-7300-261-9.

[3] Horčík, Z. *Popis rozhraní JTAG [online]*. 2004. Dostupné na: `http://noel.feld.cvut.cz/vyu/ap2/JTAGmoje.htm`.

[4] Husár, A., Přikryl, Z., Masařík, K. et al. ASIP Design using Architecture Description Language ISAC. In *ACACES 2009 - Poster Abstracts*. Ghent, BE: High Performance and Embedded Architecture and Compilation, 2009. S. 137–139. Dostupné na: `http://www.fit.vutbr.cz/research/view%5Fpub.php?id=8984`. ISBN 978-90-382-1467-2.

[5] Lissom. *Lissom Project [online]*. duben 2011. Dostupné na: `http://www.fit.vutbr.cz/research/groups/lissom/project.html`.

[6] Vašíček, Z. *Úvod – FITKit [online]*. duben 2011. Dostupné na: `http://merlin.fit.vutbr.cz/FITkit/`.

[7] Wikipedia. *Joint Test Action Group — Wikipedia, The Free Encyclopedia*. 2011. [cit. 2011-04-25]. Dostupné na: `http://cs.wikipedia.org/wiki/Joint%5FTest%5FAction%5FGroup`.