VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY

DEPARTMENT OF MICROELECTRONICS

VERIFIKACE INTEGROVANÉHO OBVODU S PROCESOREM ARM CORTEX M0/M0+

VERIFICATION OF SOC WITH ARM CORTEX M0/M0+ PROCESSOR

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE | Artem Gumenyuk

AUTHOR

VEDOUCÍ PRÁCE | doc. Ing. Lukáš Fujcik, Ph.D.

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Mikroelektronika a technologie**

Ústav mikroelektroniky

Student: Artem Gumenyuk ***ID:*** 228619
Ročník: 3 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Verifikace integrovaného obvodu s procesorem ARM Cortex M0/M0+

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s architekturou procesoru ARM Cortex M0/M0+ a jeho integrací do obvodu na úrovni RTL. Navrhněte konfigurovatelné komponenty pro ověření funkčnosti a debuggování integrovaných obvodů s procesorem ARM Cortex M0/M0+. Tyto komponenty budou monitorovat základní běh programu a kontrolovat přístup do paměti a k periferiím. Realizujte navrhnuté komponenty v jazyku SystemVerilog s využitím UVM (Universal Verification Methodology). Jejich funkčnost demonstrujte sestavením jednoduchého verifikačního prostředí, které bude tyto komponenty využívat.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 1.6.2023

Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

doc. Ing. Pavel Šteffan, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt

Práce se zabývá teorií činnosti procesoru ARM Cortex-M0/M0+ a analyzuje možnosti návrhu verifikačních komponent. Popisuje návrh a realizaci verifikačních komponent pro monitorování správné činnosti systému s procesorem ARM-CortexM0/M0+ a zobrazení důležitých údajů po celou dobu běhu simulace. Potom demonstruje výstupy komponent na sestaveném referenčním návrhu.

Klíčová slova

Verifikační komponenty, ARM, Cortex-M0, Cortex-M0+, AHB-Lite, přerušení, verifikace, procesor.

Abstract

The work deals with the theory of operation of the ARM Cortex-M0/M0+ processor and analyzes the design possibilities of verification components. It describes the design and implementation of verification components for monitoring the correct operation of the system with an ARM-CortexM0/M0+ processor and displaying important data throughout the simulation run. It then demonstrates component outputs on an assembled reference design.

Keywords

Verification components, ARM, Cortex-M0, Cortex-M0+, AHB-Lite, interrupts, verification, processor.

Bibliografická citace

GUMENYUK, Artem. *Verifikace integrovaného obvodu s procesorem ARM Cortex M0/M0+* [online]. Brno, 2023 [cit. 2023-05-28]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/152261. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce Lukáš Fujcik.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	*Artem Gumenyuk*
VUT ID studenta:	*228619*
Typ práce:	*Bakalářská práce*
Akademický rok:	*2022/23*
Téma závěrečné práce:	*Verifikace integrovaného obvodu s procesorem ARM Cortex M0/M0+*

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucího závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: 1. června 2023

----

podpis autora

Poděkování

Děkuji doc. Ing. Lukáši Fujcikovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc, ochotu pomoci i za veškeré cenné rady při zpracování mé bakalářské práce. Dále děkuji Ing. Miloši Juhásovi ze spolupracující firmy ON Design Czech, s.r.o. za poskytnutí příležitosti ke zpracování mé práce, za trpělivost při vysvětlování, za všechny cenné připomínky k práci a za veškerou odbornou pomoc. Děkuji také své rodině a přátelům, kteří mě v průběhu psaní práce podporovali.

V Brně dne: 1. června 2023 --

podpis autora

Obsah

SEZNAM OBRÁZKŮ ... 9

SEZNAM TABULEK ... 10

ÚVOD ... 11

1. TEORETICKÁ ČÁST ... 12

1.1 ZÁKLADNÍ INFORMACE O JÁDŘE A JEHO VÝHODÁCH ... 12
1.2 ARCHITEKTURA ... 13
1.2.1 Porovnání von Neumannovy a Harvardské architektury ... 13
1.2.2 Architektura ARMv6-M ... 14
1.3 INSTRUKČNÍ SADA: THUMB ... 16
1.4 AMBA 3 AHB-LITE ROZHRANÍ ... 16
1.5 VÝJIMKY A PŘERUŠENÍ ... 18
1.6 FUNKCE PRO NÍZKOU SPOTŘEBU ENERGIE ... 20
1.7 SYSTÉMOVÉ ŘÍZENÍ ... 22

2. VLASTNÍ NÁVRH ŘEŠENÍ ... 23

2.1 KONTROLA PŘÍSTUPU DO PAMĚTI ... 23
2.2 VIZUALIZAČNÍ KOMPONENTY ... 23
2.2.1 Dekódování instrukcí ... 23
2.2.2 Vizualizace činností procesoru ... 24

3. PRAKTICKÁ ČÁST ... 25

3.1 UNIVERZÁLNÍ VERIFIKAČNÍ METODOLOGIE: UVM A UVMF ... 25
3.2 POPIS SESTAVENÉHO PROSTŘEDÍ ... 28
3.3 DEKODÉR INSTRUKCÍ ... 29
3.3.1 Rozdělení instrukcí na typy a možnost změny barev vizualizace ... 32
3.3.2 Funkce výběru instrukcí pro vizualizace ... 32
3.3.3 Funkce výpisu instrukcí v textovém formátu ... 32
3.3.4 Funkce časového omezení výpisu instrukcí ... 33
3.4 KOMPONENTA MONITOROVÁNÍ PŘERUŠENÍ ... 33
3.4.1 Možnost nastavení názvu zdroje přerušení ... 34
3.5 KOMPONENTA KONTROLY PŘÍSTUPU DO PAMĚTI ... 35
3.5.1 Funkce zvýraznění komunikace procesoru s periferií ... 36
3.6 KONFIGURACE PROSTŘEDÍ ARM_UTILS ... 37
3.6.1 Možnost vypnutí komponent ... 40
3.7 NÁVOD NA INTEGRACE KOMPONENT DO SYSTÉMU UŽIVATELE ... 40
3.8 SIMULACE V XCELIUM ... 42
3.9 REFERENČNÍ NÁVRH ... 42
3.9.1 SPI rozhraní ... 43
3.9.2 Externí přerušení ... 45
3.9.3 Systém meteorologické stanice ... 46

4. ZÁVĚR ... 49

LITERATURA ... 50

SEZNAM SYMBOLŮ A ZKRATEK 51

SEZNAM OBRÁZKŮ

Obr. 1: Koncepční schéma přechodu procesoru mezi stavy [2] ... 14
Obr. 2: AHB-Lite blokový diagram [7] ... 17
Obr. 3: Koncepční schéma obsluhy přerušení [2] ... 19
Obr. 4: Koncepční schéma obsluhy přerušení s různou prioritou ... 24
Obr. 5: Blokové schéma prostředí UVM ... 25
Obr. 6: Blokové schéma sestaveného prostředí ... 28
Obr. 7: Časový diagram zpracování instrukcí ... 29
Obr. 8: Vzhled transakcí dekodéru instrukcí v simulatoru Questa ... 30
Obr. 9: Vizualizace transakcí komponenty monitorování přerušení ... 34
Obr. 10: Lokální parametry pro nastavení barev transakcí ... 35
Obr. 11: Funkce zvýraznění komunikace procesoru s periferií v Questa ... 36
Obr. 12: Příklad nastavení konfiguračních bitů pro vypnutí komponent ... 40
Obr. 13: Úprava metody body() tridy „cm0_tbench_bench_sequence_base“ ... 42
Obr. 14: Blokové schéma referenčního návrhu s procesorem ... 43
Obr. 15: Ukázka jednoho SPI rámce ... 43
Obr. 16: Koncepční schéma periferie externího přerušení ... 45
Obr. 17: Vizualizace obsluhy přerušení po rámci SPI, meteorologická stanice ... 47
Obr. 18: Činnost komponenty dekodéru instrukcí během obsluhy přerušení, meteorologická stanice ... 48
Obr. 19: Činnost komponenty kontroly přístupu do paměti během obsluhy přerušení, meteorologická stanice ... 48

SEZNAM TABULEK

Tab. 1: Výhody a nevýhody von Neumannovy architektury vůči Harvardské architektuře [6] 13
Tab. 2: Části pasivního agenta UVM 26
Tab. 3: Části aktivního agenta UVM 27
Tab. 4: Položky výstupní transakce dekodéru instrukcí 29
Tab. 5: Signály a registry, kterých využívá dekodér instrukcí 31
Tab. 6: Podmínky pro vizualizace transakce dekodérem instrukcí 31
Tab. 7: Vstupní signály komponenty monitorování přerušení 33
Tab. 8: Položky výstupní transakce komponenty monitorování přerušení 33
Tab. 9: Položky výstupní transakce komponenty kontroly přístupu do paměti 36
Tab. 10: Proměnné třídy periph_config 37
Tab. 11: Proměnné třídy env_top_config 37
Tab. 12: Metody konfigurační třídy env_top_config 39
Tab. 13: 32bitové registry pro vysílací a přijatá data SPI 44
Tab. 14: Stavový registr SPI_STAT_REG 44
Tab. 15: Konfigurační registr SPI_CFG_REG 44
Tab. 16: Konfigurační registr externího přerušení EXT_INT_CFG_REG 45
Tab. 17: Stavový registr externího přerušení EXT_INT_STAT_REG 45
Tab. 18: Rozdělení rámce pro zápis, meteorologická stanice 46
Tab. 19: Rozdělení rámce pro čtení, meteorologická stanice 46

ÚVOD

Procesor ARM Cortex-M0/M0+ je možné integrovat do systému na čipu a použít jako hlavní řídicí jednotku nebo pro pomocnou činnost. Nedílnou součástí návrhu jsou vlastní periferie. Periferie jsou integrovány do systému a komunikují s procesorem přes společnou sběrnici. Vzhledem ke komplexitě jádra, které má velké množství funkcí a signálů, není ani s využitím moderních simulátorů jednoduché přímo pozorovat, v jakém stavu se jádro právě nachází a která část programu se vykonává. Na rozdíl od ladění návrhu s vlastním RTL vnáší procesor vykonávající program prostor pro zanesení dalších chyb, které mohou vzniknout při návrhu softwaru. Návrh takového obvodu, jeho ladění a verifikaci je možné ulehčit pomocí vhodných verifikačních komponent. Ty odhalí nejčastější chyby a pomůžou vizualizovat běh programu a stav jádra procesoru v digitálním simulátoru.

Cílem této práce je seznámit čtenáře s architekturou procesoru ARM Cortex-M0/M0+, vysvětlit, jak periferie komunikují s procesorem a jakým způsobem procesor zpracovává přerušení. Dalším cílem je navrhnout a realizovat univerzální komponenty pro ověření funkčnosti a debuggování integrovaných obvodů s procesorem ARM Cortex-M0/M0+.

V teoretické části práce jsou vysvětlené principy činností procesoru a popsána jeho architektura. Důležitým je popis komunikace s periferiemi přes sběrnici AMBA 3 AHB-Lite. V této části jsou rozebrané všechny druhy výjimek a přerušení, které se mohou vyskytovat během činnosti procesoru, a systém jejich priorit. V teoretické části jsou také popsané funkce procesoru pro nízkou spotřebu energie a jeho režimy spánku.

V části vlastního návrhu řešení jsou popsány verifikační komponenty, které byly implementovány v rámci této práce. Na realizace těchto komponent byly použity znalosti o architektuře, systémových registrech, typech přenosů na sběrnici a možných režimech a stavech, ve kterých se procesor může nacházet.

V praktické části práce je popsána základní struktura a principy Univerzální verifikační metodologie (UVM) a její rozšíření – UVMF. V této části jsou popsané principy funkce, způsob realizace a možností třech verifikačních komponent. Práce pokračuje v popisu konfigurace prostředí s těmito komponenty a návodem na integraci tohoto prostředí do prostředí uživatele. Potom následuje kapitola o realizaci skriptu pro simulaci v dalším digitálním simulátoru Xcelium. Práce je ukončena popisem referenčního návrhu pro demonstraci činností realizovaných verifikačních komponent.

1. TEORETICKÁ ČÁST

1.1 Základní informace o jádře a jeho výhodách

ARM Cortex-M0 je 32bitový procesor určený pro širokou škálu vestavěných aplikací. Vývojářům nabízí významné výhody, včetně [1]:

- Jednoduché architektury, která se snadno učí a programuje;
- Velmi nízké spotřeby energie a energeticky účinného provozu;
- Vynikající hustoty kódu[1];
- Deterministického, vysoce výkonného zpracování přerušení;
- Vzestupné kompatibility s rodinou procesorů Cortex-M.

Cortex-M0 je procesor s vysokou optimalizací plochy a spotřeby, s třístupňovou pipeline[2] von Neumannovy architektury. Cortex-M0 je nejmenší procesor v celé rodině Cortex-M, v minimální konfiguraci obsahuje kolem 12000 logických hradel. Procesor implementuje architekturu ARMv6-M, která je založena na 16bitové instrukční sadě Thumb a zahrnuje technologii Thumb-2. Ta poskytuje výjimečný výkon, který se očekává od moderní 32bitové architektury, s vyšší hustotou kódu než jiné 8bitové a 16bitové mikrokontroléry. Efektivní provádění kódu umožňuje pomalejší takt procesoru nebo zvýšení doby režimu spánku, což snižuje celkovou spotřebu [2].

Řešení poskytuje výjimečnou energetickou účinnost díky malé, ale výkonné instrukční sadě a optimalizovanému návrhu. Taký poskytuje výpočetní hardware a jednocyklovou násobičku [2]. Pomocí instrukce MULS je možné provést násobení dvou 32bitových operandů, přičemž výsledek je také 32bitový [3].
Cortex-M0/M0+ úzce integruje konfigurovatelné vnořené vektorové přerušení (Nested Vectored Interrupt) [2].
Kontrolér přerušení (NVIC) poskytuje výborný výkon v oblasti přerušení [2]:

- Obsahuje nemaskovatelné přerušení;
- Poskytuje možnost přerušení s nulovým chvěním[3];
- Poskytuje čtyři úrovně priority přerušení.

Těsná integrace jádra procesoru a NVIC zajišťuje rychlé provádění procedur obsluhy přerušení, což výrazně snižuje latenci. Pro optimalizaci návrhů s nízkou spotřebou

---

[1] Vynikající hustota kódu znamená, že pro vykonání stejné úlohy potřebuje procesor provést menší počet instrukcí než v případě méně pokročilé instrukční sady.
[2] Pipeline umožňuje zpracovávat jednotlivé úlohy paralelně a nečekat na dokončení jedné úlohy a pak přejít na další. Ve výpočetní technice označuje pipeline logickou frontu, která je naplněna všemi instrukcemi, jež má počítačový procesor paralelně zpracovávat [4].
[3] Chvění je rozptyl zpoždění přerušení v nejlepším a nejhorším případě.

energie je NVIC integrován s režimy spánku. Tyto režimy zahrnují funkci hlubokého spánku, což umožňuje rychlé vypnutí celého zařízení [2].

1.2 Architektura

1.2.1 Porovnání von Neumannovy a Harvardské architektury

Ve von Neumannově architektuře se stejná paměť a sběrnice používají k ukládání dat i instrukcí, které spouštějí program [5].

Harvardská architektura ukládá strojové instrukce a data do samostatných paměťových jednotek, které jsou propojeny různými sběrnicemi. V tomto případě je třeba pracovat minimálně se dvěma adresovými prostory paměti, takže existuje paměťový registr pro strojové instrukce a další paměťový registr pro data. Počítače navržené s harvardskou architekturou jsou schopny spouštět program a přistupovat k datům nezávisle, a tedy současně. Harvardská architektura striktně odděluje data a kód [5].

Tab. 1: Výhody a nevýhody von Neumannovy architektury vůči Harvardské architektuře [6]

Výhody von Neumannovy architektury	Nevýhody von Neumannovy architektury
Řídicí jednotka načítá instrukce a data stejným způsobem z jedné paměťové jednotky. To zjednodušuje vývoj a konstrukci řídicí jednotky.	Paralelní spouštění programů není povoleno kvůli sériovému zpracování instrukcí.
Výše uvedená výhoda taký znamená, že k datům z paměti a ze zařízení se přistupuje stejným způsobem. Tím se zvyšuje efektivita.	V jednom okamžiku lze přistupovat pouze k jedné sběrnici. To vede k tomu, že procesor je nečinný (protože je rychlejší než datová sběrnice). Toto je považováno za von Neumannovo úzké hrdlo.
Programátoři mají kontrolu nad organizací paměti.	Uložení instrukcí a dat na stejném místě lze považovat za výhodu jako celek. To však, v případě chyby v programu, může mít za následek přepisování pamětí, což vede ke ztrátě dat.

Jak bylo uvedeno v **Tab. 1**, von Neumannova architektura má tzv. úzké hrdlo. Důvodem je to, že procesor nevykonává žádnou užitečnou práci, zatímco čeká na načtení dat z paměti. To znamená, že jestli procesor je činný, ale rychlost přenosu je malá, tak odezva celého systému bude určena rychlostí přenosu. Způsobů, jak řešit tento problém, je několik, mezi nejčastěji používané patří [6]:

Ukládání do mezipaměti

Data, která se často používají, se ukládají do paměti s rychlejším přístupem. Tato paměť je často menší a dražší než hlavní paměť.

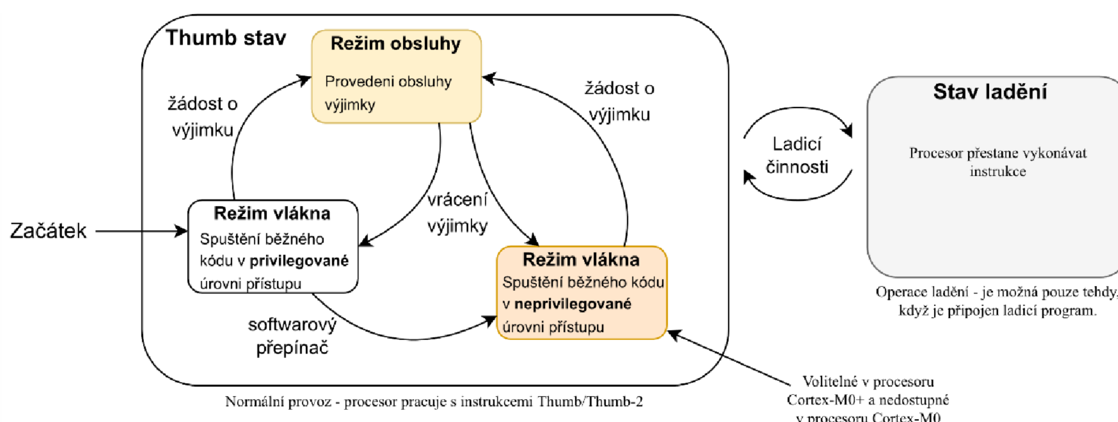
Přednačítání

Přenos některých dat do mezipaměti předtím, než jsou požadována. Tím se urychlí přístup v případě požadavku na data.

Vícevláknové zpracování
Spravování několika požadavků najednou v samostatných vláknech.
Nové typy paměti RAM
Například paměť DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory), ze které se dají vyčítat data na obě hrany systémových hodin.
Zpracování v paměti
Integrace procesoru a paměti na jednom čipu.

Pro řešení problému úzkého hrdla používá ARM Cortex-M0 třístupňovou pipeline (paralelní zpracování) a ukládání do registru uvnitř jádra.

1.2.2 Architektura ARMv6-M



Obr. 1: Koncepční schéma přechodu procesoru mezi stavy [2]

Procesory ARM Cortex-M0 a Cortex-M0+ jsou založeny na architektuře ARMv6-M. Ne vše je v definici architektury ARMv6-M pevně stanoveno. Některé funkce definované v architektuře mohou být volitelné. Například jednotka ochrany paměti (MPU) je volitelná a počet zdrojů přerušení podporovaných v zařízení mohou konfigurovat návrháři čipů. Některé oblasti architektury mohou být definovány implementací. Například počet hodinových cyklů pro provedení instrukce je specifický pro návrh procesoru. Podobně je tomu i u počtů identifikačních (ID) registrů [2].

Architektura ARMv6-M má dva provozní režimy a dva stavy. Kromě toho může mít privilegované a neprivilegované úrovně přístupu. Privilegovaná úroveň přístupu může přistupovat ke všem prostředkům v procesoru, zatímco neprivilegovaná úroveň znamená, že některé oblasti paměti mohou být nepřístupné a několik operací nelze použít. Úroveň neprivilegovaného přístupu není v procesoru Cortex-M0 k dispozici a je volitelná (specifická pro zařízení) v procesoru Cortex-M0+ [2]. Koncepční schéma přechodu mezi stavy je uvedená na obrázku 1.

Když procesor spouští program, je ve stavu Thumb. V tomto stavu může být buď v režimu vlákna (Thread mode), nebo v režimu obsluhy přerušení (Handler mode). V architektuře ARMv6-M jsou režimy Thread a Handler téměř zcela shodné. Jediný rozdíl je v tom, že režim vlákna může používat tzv. zastíněný ukazatel zásobníku nastavením speciálního registru CONTROL [2].

Thumb stav je normální stav práce s instrukcemi. Debug je stav ladění (Debug state) a je aktivní, když je procesor zastaven například ladícím programem prostřednictvím příkazu ladění [2].

Procesory Cortex-M0 a Cortex-M0+ mají k dispozici banku registrů s šestnácti 32bitovými registry. Většina z nich je obecná, registry R13 - R15 mají speciální účely. R13 je ukazatel zásobníku (SP), R14 – registr odkazů (LR), R15 – čítač instrukcí (PC). Existují 3 speciální registry: PSR – stavový registr programu, PRIMASK – registr masky priorit, CONTROL – registr nastaveného ukazatele zásobníku. PSR zahrnuje v sobě 3 registry: APSR – obsahuje aktuální stav příznaků předchozí provedené instrukce, IPSR – obsahuje číslo výjimky, kterou procesor aktuálně obsluhuje, EPSR – stavový registr Thumb [1].

Všechny procesory ARM Cortex-M mají 4 GB paměťového adresního prostoru. Tento paměťový prostor je sdílený mezi instrukční pamětí (ROM), datovou pamětí (RAM), periferiemi, vestavěnými periferiemi procesoru (např. řadičem přerušení (NVIC)) a pamětí ladící komponenty procesoru. Paměťový systém v procesorech Cortex-M podporuje paměťové přenosy různých velikostí, jako jsou bajty (8 bitů), půlslova (16 bitů) a slova (32 bitů). Cortex-M0 a Cortex-M0+ lze nakonfigurovat tak, aby podporovaly paměťové systémy little endian nebo big endian, ale není možné přepínat z jednoho na druhý v rámci jedné implementace [2].

Aby bylo možné upřednostnit požadavky na přerušení a zpracovat další výjimky, procesory Cortex-M mají v sobě integrovaný řadič přerušení nazvaný NVIC. Funkce správy přerušení je řízena řadou programovatelných registrů v NVIC [2].

NVIC podporuje řadu funkcí jako například: flexibilní správa přerušení, podpora vnořených přerušení, vektorové zadávání výjimek a maskování přerušení [2].
Přestože jsou procesory Cortex-M0 a Cortex-M0+ v současné době nejmenšími procesory z rodiny procesorů ARM, podporují řadu ladících funkcí. Procesor umožňuje ladění v režimu zastavení, krokové ladění, přístupy k registrům a přístupy do paměti. Pro ladění poskytují funkce, jako například [2]:

1) Jednotka přerušení (Breakpoint Unit) poskytuje až čtyři breakpointy s hardwarovou podporou. Hardwarový komparátor porovnává aktuální adresy

programu s referenční adresou nastavenou ladícím programem. Když procesor načte a provede instrukci z této adresy, komparátor vygeneruje signál události ladění, který zastaví procesor [2].

2) Datový sledovací bod (Data Watchpoint), který podporuje až dva body sledování. Datovou nebo periferní adresu lze označit jako sledovanou proměnnou a přístup k této adrese způsobí vygenerování ladící události, která zastaví provádění programu [2].

1.3 Instrukční sada: Thumb

Instrukční sada podporovaná procesory ARM Cortex-M se nazývá Thumb, přičemž Cortex-M0 a Cortex-M0+ podporují pouze podmnožinu definovaných instrukcí (56 z nich) [2].

Většina z těchto instrukcí má velikost 16 bitů, ale procesory Cortex-M0/M0+ podporují také šest 32bitových Thumb instrukcí z řady technologie Thumb-2. S takto malou sadou instrukcí nejsou procesory Cortex-M0 a Cortex-M0+ určeny pro náročné úlohy zpracování čísel. Procesory Cortex-M0 a Cortex-M0+ jsou navrženy pro obecné zpracování dat, řízení vstupů a výstupů a pro systémy s velmi nízkou spotřebou a nízkými náklady, kde je třeba, aby velikost čipu byla malá [2].

Jednou z klíčových charakteristik instrukční sady pro procesory Cortex-M je vzestupná kompatibilita. Programový kód vyvinutý pro procesory Cortex-M0 a Cortex-M0+ lze často spustit na procesorech Cortex-M3, Cortex-M4 a Cortex-M7 beze změny [2].

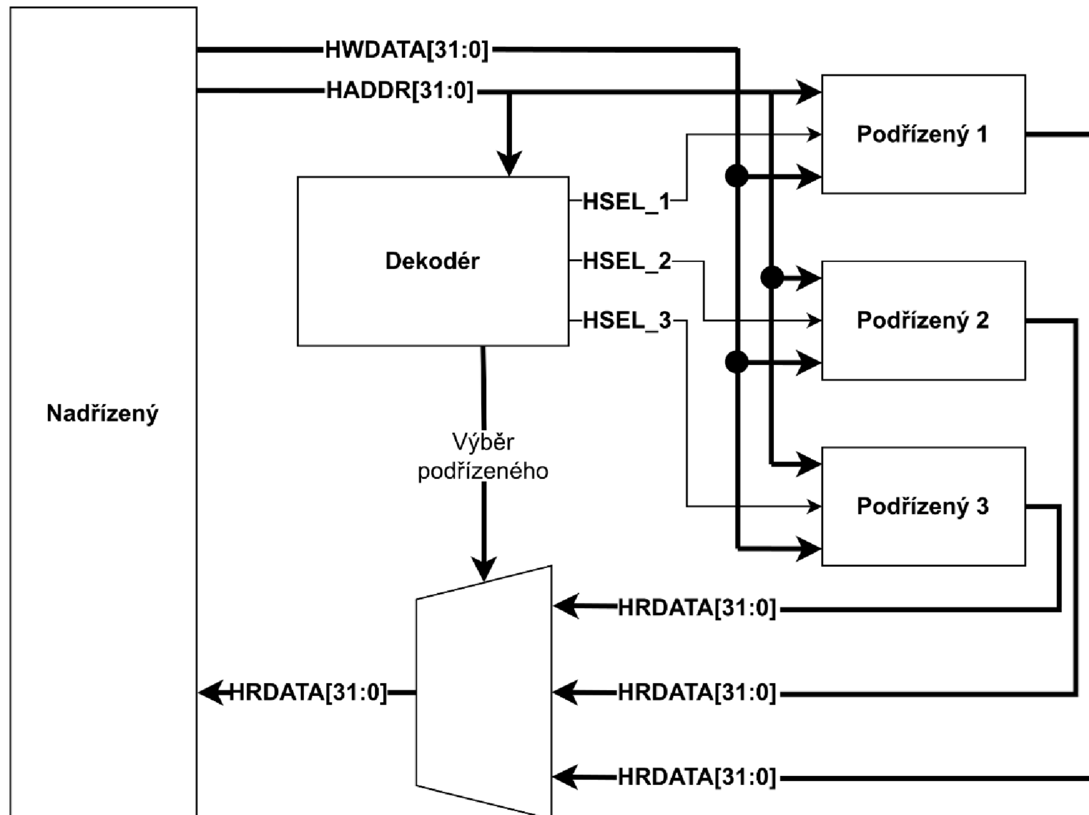
1.4 AMBA 3 AHB-Lite rozhraní

Procesor provádí veškerou komunikaci s externími pamětmi a zařízeními prostřednictvím rozhraní AHB-Lite. AMBA AHB-Lite splňuje požadavky na vysoce výkonné syntetizovatelné návrhy. Jedná se o sběrnicové rozhraní, které podporuje jedinou nadřízenou jednotku a zajišťuje provoz s vysokou šířkou pásma [7].

AHB-Lite implementuje funkce potřebné pro vysoce výkonné systémy s vysokou taktovací frekvencí, včetně:

- Sériových přenosů;
- Provozu s jednou taktovací hranou;
- Netřístavových implementací;
- Širokých konfigurací datové sběrnice: 32, 64, 128, 256, 512 a 1024 bitů [7].

Obrázek 2 ukazuje návrh systému AHB-Lite s jednou nadřízenou jednotkou a třemi podřízenými jednotkami. Logika propojení sběrnice se skládá z jednoho dekodéru adres a multiplexoru slave-to-master. Dekodér monitoruje adresu z nadřízené jednotky tak, aby byl vybrán příslušná podřízená jednotka, a multiplexor směřuje odpovídající výstupní data vybraného podřízeného zpět do nadřízené jednotky [7].



Obr. 2: AHB-Lite blokový diagram [7]

AMBA AHB-Lite interface je velmi důležitou částí procesoru, a proto je nutné plně pochopit jeho funkci.

Nadřízená jednotka zahájí přenos řízením adresního a řídicího signálu. Tyto signály poskytují informace o adrese, směru a šířce přenosu a udávají, zda je přenos součástí série.

Datová sběrnice pro zápis přenáší data z nadřízeného zařízení do podřízeného a datová sběrnice pro čtení přenáší data z podřízeného zařízení do nadřízeného.

Každý přenos se skládá z [7]:

- Adresní fáze: jeden adresní a řídicí cyklus;
- Datové fáze: jeden nebo více cyklů pro data.

Podřízená jednotka nemůže požádat o prodloužení adresní fáze, a proto musí být všechny podřízené jednotky schopny během této doby odebírat vzorky adresy. Podřízená jednotka však může požadovat, aby nadřízená jednotka prodloužila datovou fázi pomocí

HREADY. Pokud je tento signál v nule, způsobí, že do přenosu vloží čekací stavy a umožní podřízené jednotce získat dodatečný čas na poskytnutí nebo vzorkování dat. Podřízená jednotka používá HRESP k indikaci úspěchu nebo neúspěchu přenosu [7].

Přenosy lze rozdělit do čtyř typů, které se řídí pomocí signálu **HTRANS**[1:0]:

HTRANS[1:0] = b00 **IDLE**(„nečinný“)
Označuje, že není vyžadován žádný přenos dat. Nadřízená jednotka používá přenos IDLE, pokud nechce provést přenos dat. Doporučuje se, aby nadřízená jednotka ukončila uzamčený přenos přenosem IDLE. Podřízené zařízení musí vždy poskytnout odpověď „OKAY“ ve stavu čekání na přenos IDLE a musí přenos ignorovat [2].

HTRANS[1:0] = b01 **BUSY** („zaměstnaný“)
Typ přenosu BUSY umožňuje nadřízené jednotce vkládat IDLE cykly doprostřed série přenosů. Tento typ přenosu signalizuje, že nadřízená jednotka pokračuje v sérii, ale další přenos se nemůže uskutečnit okamžitě. Když nadřízená jednotka použije typ přenosu BUSY, musí adresní a řídicí signály odrážet další přenos v sérii. Pouze série nedefinované délky mohou mít jako poslední cyklus série přenos BUSY. Podřízené zařízení musí vždy poskytnout odpověď „OKAY“ ve stavu čekání na přenos BUSY a musí přenos ignorovat [2].

HTRANS[1:0] = b10 **NONSEQ**(„ne sekvenční“)
Indikuje samostatný přenos nebo první přenos série. Adresní a řídicí signály nesouvisí s předchozím přenosem. Jednotlivé přenosy na sběrnici jsou považovány za série o délce jedna, a proto je typ přenosu NONSEQUENTIAL [2].

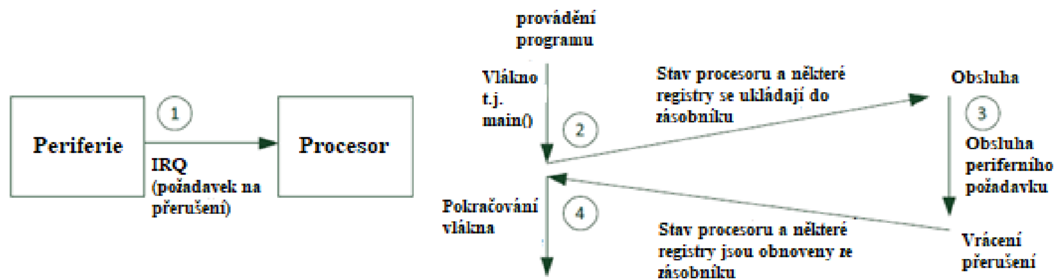
HTRANS[1:0] = b11 **SEQ**(„sekvenční“)
Zbývající přenosy v sérii jsou sekvenční a adresa se vztahuje k předchozímu přenosu. Řídicí informace jsou shodné s předchozím přenosem. Adresa se rovná adrese předchozího přenosu plus velikost přenosu v bajtech, přičemž velikost přenosu je signalizována signály HSIZE[2:0]. V případě zalamování při dosažení horní hranice adresy se adresa zabalí na nižší adresu [7].

1.5 Výjimky a přerušení

Ve většině mikrokontrolérů umožňuje funkce přerušení perifernímu zařízení nebo externímu hardwaru odeslat požadavek procesoru (IRQ), aby procesor mohl provést část kódu, která má za úkol vyřídit tento požadavek [2].

Tento proces zahrnuje pozastavení aktuálně prováděné úlohy nebo probuzení z režimu spánku a provedení části softwarového kódu nazývané obsluha výjimky, která

požadavek obslouží. Po obsloužení požadavku může procesor pokračovat v předchozí přerušené práci s kódem [2]. Koncepční schéma práce s přerušením je na obrázku 3.



Obr. 3: Koncepční schéma obsluhy přerušení [2]

Obecně je přerušení jen jedním z typů výjimek v procesorech ARM Cortex-M. Procesory Cortex-M0 a Cortex-M0+ obsahují vestavěný řadič přerušení NVIC, který podporuje až 32 vstupů IRQ (Interrupt request), vstup NMI (Non-Maskable Interrupt) a řadu systémových výjimek uvnitř procesoru. V závislosti na konstrukci mikrokontroléru mohou být IRQ a NMI generovány buď z periferií na čipu, nebo z externích zdrojů [2].

Každý zdroj výjimky v procesoru Cortex-M0 nebo Cortex-M0+ má jedinečné číslo výjimky. Číslo výjimky pro NMI je 2 a čísla výjimek pro periferie na čipu a externí zdroje přerušení jsou od 16 do 47. Ostatní čísla výjimek od 1 do 15 jsou určena pro systémové výjimky generované uvnitř procesoru, přičemž některá čísla výjimek v tomto rozsahu se nepoužívají [2].

Každý typ výjimky má také přiřazenou prioritu. Úrovně priority některých výjimek jsou pevné a některé jsou programovatelné. Pevné úrovně jsou od -3 do -1, kde -3 je nejvyšší možná priorita. Používají se pro výjimky jako: Reset (-3), NMI (-2), Hard Fault (-1). Programovatelné úrovně jsou od 0 do 3, kde 3 je nejmenší priorita. Jeden ze čtyř úrovní priority dá v programu přiřadit libovolné výjimce nebo přerušení, kromě výjimek s pevně nastavenou prioritou [2].

Latence přerušení je počet hodinových cyklů od výskytu výjimky do začátku její obsluhy, tj. vykonání části kódu odpovídajícího přerušení. Pro Cortex-M0 je latence 16 hodinových cyklů a 15 pro Cortex-M0+. Procesor dovoluje nastavit latenci IRQ na větší počet hodinových cyklů pomocí signálu připojeného k NVIC. Procesor zaručuje obsluhu přerušení s nulovým chvěním, to znamená, že latence bude vždy konstantní [2].

Typy výjimek v procesorech Cortex-M0 a Cortex-M0+:

Nemaskovatelné přerušení (NMI)

NMI je podobné IRQ, ale nelze jej zakázat a má nejvyšší prioritu kromě resetu. Je velmi užitečné pro bezpečnostně kritické systémy, jako je průmyslové řízení nebo automobilový průmysl [2].

Těžká závada (HardFault)

HardFault je typ výjimky určený k ošetření chybových stavů během provádění programu. Těmito chybovými stavy mohou být pokus o vykonání neznámého opkódu, chyba na rozhraní sběrnice nebo paměťového systému nebo nelegální operace, jako je pokus o přepnutí do režimu ARM [2].

Volání vedoucího (SVCall)

Výjimka SVCall nastane při provedení instrukce SVC. SVC se obvykle používá v systému s operačním systémem (OS) a umožňuje aplikacím přístup k systémovým službám [2].

Závazné servisní volání (PendSV)

Další výjimkou pro aplikace s OS je závazné servisní volání (PendSV). Na rozdíl od výjimky SVCall, která se musí spustit okamžitě po provedení instrukce SVC, lze výjimku PendSV odložit. PendSV se běžně používá v OS k plánování systémových operací, které se mají provést až po dokončení úloh s vyšší prioritou [2].

Časovač systémových tiků (SysTick)

Časovač SysTick uvnitř NVIC je volitelnou funkcí pro aplikaci OS. Téměř všechny OS potřebují časovač pro generování pravidelných přerušení pro údržbu systému, jako je přepínání kontextu [2].

Přerušení (IRQ – Interrupt Request)

Externí přerušení je třeba před použitím povolit. Pokud přerušení není povoleno, nebo pokud již procesor používá jiný obslužný program pro výjimky se stejnou nebo vyšší prioritou, IRQ se uloží do registru čekajících stavů [2].

1.6 Funkce pro nízkou spotřebu energie

Procesory Cortex-M obecně obsahují následující funkce pro nízkou spotřebu energie [2]:

Dva architekturní režimy spánku

Normální spánek a hluboký spánek. Režimy spánku lze dále rozšířit o funkce řízení rychlosti specifické pro výrobce. V rámci procesoru se oba režimy spánku chovají podobně. Zbytek mikrokontroléru však může obvykle snížit spotřebu použitím různých úrovní metod snižování spotřeby specifických pro zařízení na základě těchto dvou režimů [2].

Dvě instrukce pro vstup do režimů spánku

WFE (čekání na událost) a WFI (čekání na přerušení). Obě lze použít s režimy normálního a hlubokého spánku [2].

Funkce spánek při ukončení (Sleep-On-Exit)

Jednou z funkcí procesorů Cortex-M pro nízkou spotřebu je funkce spánek při ukončení (Sleep-On-Exit). Je-li tato funkce povolena, procesor automaticky přejde do režimu spánku WFI při ukončení obsluhy výjimky, a pokud na zpracování nečeká žádná další výjimka. Tato funkce je užitečná pro aplikace, kde jsou činnosti procesoru řízené přerušením [2].

Volitelná funkce řadič probouzení (WIC)

Umožňuje úplné vypnutí hodin procesoru během spánku. Při použití této funkce s technologií udržování stavu, která se vyskytuje v některých moderních procesech křemíkové implementace, může procesor přejít do stavu vypnutí s extrémně nízkým únikem energie, a přesto je schopen se téměř okamžitě probudit a pokračovat v činnosti. Když je detekováno přerušení, WIC odešle požadavek jednotce řízení spotřeby (PMU) uvnitř mikrokontroléru, aby obnovila napájení a taktovací signály do procesoru, a pak se procesor může probudit, pokračovat v činnosti a zpracovat požadavek na přerušení [2].

Implementace návrhu s nízkou spotřebou energie: k co největšímu snížení spotřeby energie byly použity různé návrhové techniky. Protože počet hradel je také velmi nízký, je statický unikající výkon procesoru ve srovnání s většinou ostatních 32bitových mikrokontrolérů nepatrný [2]. Ke snížení spotřeby navíc přispívají i různé vlastnosti procesorů Cortex-M:

Vysoký výkon

Výkon procesorů Cortex-M0 a Cortex-M0+ je často několikanásobně vyšší než u mnoha populárních 8bitových/16bitových mikrokontrolérů. Díky tomu lze stejné výpočetní úlohy provést za kratší dobu a mikrokontrolér může zůstat delší dobu v režimu spánku. Alternativně může mikrokontrolér běžet na pomalejší taktovací frekvenci, aby provedl stejnou požadovanou výpočetní úlohu a snížil tak spotřebu energie [2].

Vysoká hustota kódu

Díky velmi efektivní instrukční sadě lze snížit požadovanou velikost programu a v důsledku toho lze použít mikrokontrolér založený na Cortex-M0 nebo Cortex-M0+ s menší pamětí flash a snížit tak spotřebu energie a náklady [2].

Metoda snižování výkonu během spánku obvykle zahrnuje následující [2]:

- Zastavení některých nebo všech hodinových signálů;
- Snížení taktovací frekvence některých logických obvodů;
- Snížení napětí na různých částech mikrokontroléru;
- Vypnutí napájení některých částí mikrokontroléru.

Do režimů spánku lze vstoupit třemi různými způsoby:

- Provedením instrukce WFE (čekání na událost);
- Provedením instrukce WFI (čekání na přerušení);
- Použitím funkce spánek při ukončení (Sleep-On-Exit).

1.7 Systémové řízení

Uvnitř adresového rozsahu SCS (0xE000E000 až 0xE000EFFF) je v procesorech Cortex-M zabudována řada řídicích registrů. Patří mezi ně vnořené registry Vektorového řadiče přerušení (NVIC) pro správu přerušení, řada registrů System Control Block (SCB) pro řízení systému včetně správy funkcí režimu spánku, registry nastavení časovače SysTick a registry jednotky ochrany paměti (MPU) – programovatelná jednotka pro řízení oprávnění přístupu do paměti a atributů paměti. Funkce MPU je volitelná funkce, která je k dispozici u procesoru Cortex-M0+ a není k dispozici u procesoru Cortex-M0 [2].

V řadě registrů SCB je hodně registrů, které je možné použít např. v konfigurovatelných verifikačních komponentách:

Základní registr CPU ID – obsahuje informaci o čísle procesoru a o jeho revizi;

SHPRx registry – obsahují informaci o prioritě výjimek SVC, SysTick, PendSV;

ICSR – stavové a řídicí registry přerušení;

SCR – registry režimů spánku a registr čekání na událost.

2. VLASTNÍ NÁVRH ŘEŠENÍ

V této části budou popsané návrhy vlastních verifikačních komponent, které budou monitorovat běh programu, zobrazovat informace o aktuálně prováděných operacích během celé doby simulace a částečně kontrolovat jejich správnost.

2.1 Kontrola přístupu do paměti

Základem této komponenty je kontrola přístupu procesoru do pamětí periferií. V případě přístupu do nesprávné oblasti pamětí komponenta musí vyhlásit varování o chybné transakci a zobrazit o ní informaci.

Důležitou částí práce každé periferie je čtení a ukládání dat do jednotlivých registrů v přidělené oblasti pamětí pro tuto periferii. Velikost pamětí a rozsah adres jsou definované návrhářem a jsou dané počtem a velikostmi registrů. V ARM Cortex-M0 je přiděleno 512 MB paměti pro periferie v rozsahu adres od 0x4000_0000 do 0x5FFF_FFFF. V okamžiku, kdy procesor přečte, dekóduje a vykoná instrukci o čtení/zápisu do registru jednotlivé periferie, vygeneruje 32bitovou adresu, kterou pošle dekodéru AMBA AHB-Lite komunikační sběrnice, kde podle části adresy od MSB bude určeno, do které periferie chce procesor přistoupit. Zbývající část adresy bude zaslána do vybrané periferie, kde podle ní bude řešeno, s jakým registrem chce procesor pracovat.

Adresy jednotlivých periferií a jejich registrů budou definované návrhářem v tzv. konfigurační databázi. Komponenta bude porovnávat adresy zasílané procesorem přes sběrnici a adresy uložené v konfigurační databázi, cílem bude detekovat přístup procesoru mimo definovaný rozsah adres. Zasílání nesprávné adresy může být způsobeno chybou v programu, chybou návrháře při definování paměťové mapy nebo nesprávnou definicí adres v dekodéru sběrnice.

2.2 Vizualizační komponenty

2.2.1 Dekódování instrukcí

Cílem této komponenty bude dekódování instrukcí v reálném čase během simulace.

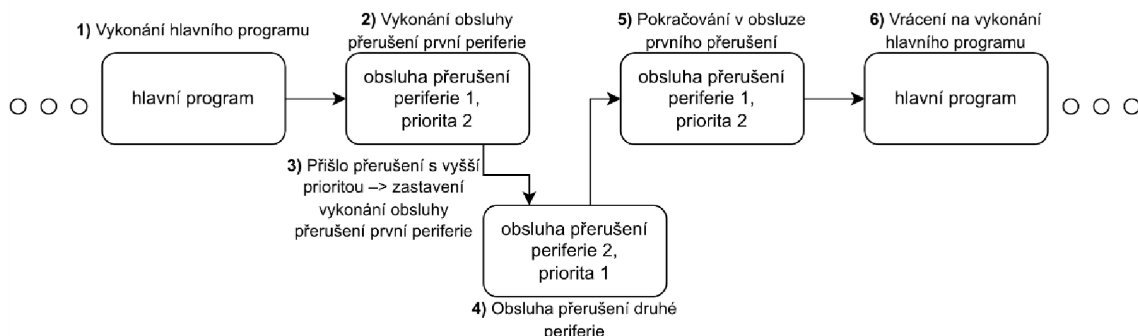
Hodnota opkódu bude dekódována a převedena do podoby assembler instrukcí. Možnosti zobrazení aktuálně vykonávaných instrukcí jsou dvě: umístění do výstupního signálu, který potom bude možné zobrazit v grafickém rozhraní s průběhy dalších signálů, druhou možností zobrazení je vypsání instrukcí textově do terminálu. Pro grafické zobrazení informace budou použité systémové funkce simulátoru. Textové zobrazení instrukcí bude doplněno údaji o časových intervalech jejich vykonání.

2.2.2 Vizualizace činností procesoru

Účelem této komponenty bude monitorovat práci procesoru, respektive zobrazovat důležitou informaci o aktuálně prováděné části programu.

Komponenta musí rozeznat výskyt přerušení, začátek a konec jeho obsluhy. Komponenta bude schopna rozeznat příchod přerušení s vyšší prioritou a označit stávající přerušení jako „nedokončené“.

Pro detekce vzniku přerušení periferií bude komponenta využívat porty IRQ a NMI. Monitorování vykonávání obsluhy přerušení bude uskutečněno pomocí informace z čítače instrukcí, registru odkazů (Link Register – LR) a z dalších systémových registrů. Koncepční vzhled výstupu komponenty v grafickém rozhraní simulačního programu je uveden na obrázku 4.



Obr. 4: Koncepční schéma obsluhy přerušení s různou prioritou

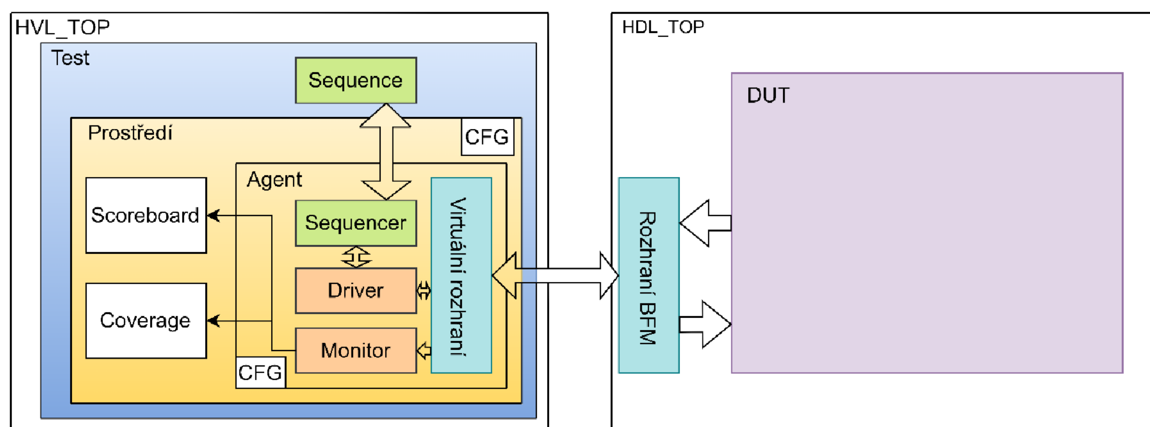
3. PRAKTICKÁ ČÁST

V této kapitole bude popsána realizace verifikačních komponent, jejich rozhraní, princip funkce a postup integrace komponent do prostředí. Komponenty jsou realizované v jazyce SystemVerilog s využitím verifikační metodologie UVMF.

3.1 Univerzální verifikační metodologie: UVM a UVMF

Univerzální verifikační metodologie (UVM) je standard, který umožňuje rychlejší realizaci verifikačního prostředí, zvyšuje úroveň př jednotlivých částí a zavádí přesnou strukturu celého verifikačního prostředí.

UVM využívá principy objektově orientovaného programování. Celé prostředí se skládá ze tříd, které jsou propojené mezi sebou a vzájemně si vyměňují data. Třídy implementují metody, které vykonávají potřebné operace. Součástí UVM je sada základních tříd, od kterých se odvádí další třídy, jako například agent, monitor, driver atd. Struktura UVM je ukázána na obrázku 5. Skládá se hierarchicky z hlavní jednotky HVL_TOP (Testbench), uvnitř které se nachází test, který může být snadno vyměněn dle potřeby návrháře. Uvnitř testů se nachází libovolný počet prostředí (Environment), uvnitř kterých se může nacházet další prostředí (Subenvironment) nebo agenti, kteří vykonávají zadané funkce. Náplň agentů volí návrhář, ale základními částmi mohou být: virtuální interface, monitor, driver, sequencer atd. Agent komunikuje se samotným návrhem prostřednictvím párů virtuálního rozhraní – rozhraní BFM. Rozhraní BFM interaguje s návrhem na úrovni signálů a předává virtuální odkáz (handle) konfigurační databázi, který se potom posílá virtuálnímu rozhraní [8]. Rozhraní BFM a samotný návrh jsou umístěné v části HDL_TOP. Struktura se dvěma vrchlovými jednotkami není jedinou variantou sestavení projektu.



Obr. 5: Blokové schéma prostředí UVM

UVM zavádí mnoho možností konfigurací pro každou část systému. Předávání konfigurace v UVM může být uskutečněno pomocí konfiguračních objektů. Konfigurační objekty se předávají mezi úrovní systému pomocí konfigurační databáze. Pomocí konfiguračních objektů se dá zapnout/vypnout vytvoření instance nebo změnit chování komponenty. Předávání odkázů (handlerů) virtuálních rozhraní mezi HVL částí a HDL částí je provedeno pomocí konfiguračních objektů. Pomocí konfiguračních objektů lze konfigurovat sekvence, třeba vytvořit chybnou sekvenci pro testování chování návrhu.

UVM požívá fáze k uspořádání hlavních kroků, které probíhají během simulace. Této fáze lze rozdělit do tří hlavních skupin: fáze sestavení, fáze běhu simulace, ukončovací fáze. Během fáze sestavení (build phase, connect phase, end of elaboration phase) se konfiguruje a sestavuje testbench. Za fáze běhu (start of simulation, run phase) se vykonavájí všechny testy. Ukončovací fáze (extract, check, report and final phases) slouží ke shromažění výsledků a jejích hlášení [11].

UVM Framework je balík s otevřeným zdrojovým kódem. UVMF rozšiřuje UVM o nové základní třídy, zavádí přesnější strukturu testbenche a má pozitivní vliv na opakované využití komponent. UVMF má generátor kódu, který je schopen rychle vygenerovat celý testbench podle popisu v jazyce YAML. Často, YAML soubory se rozdělují na tři hlavní typy: testbench, prostředí, rozhraní. Popis rozhraní určuje vstupní/výstupní signály, transakční proměnné, vlastní datové typy, parametry, konfigurační proměnné agentů a další nastavení. U popisu transakce lze stanovit její datový typ, jestli se transakce bude porovnávat, třeba v scoreboard[4], nebo jestli je potřeba danou transakci naplňovat náhodnými daty pro účely funkčního pokrytí. Pomocí YAML popisu se dá přidávat libovolný počet prostředí, sub-prostředí, agentů. Agenti mohou být aktivního nebo pasivního charakteru, v závislosti nastaveních agentů v souboru popisu testbenche. Části, které mohou být uvnitř pasivního agenta jsou v tabulce 2. Části aktivního agenta jsou v tabulce 3.

Tab. 2: Části pasivního agenta UVM

Název	**Popis**
Monitor	Vzorkuje signály DUT prostřednictvím virtuálního rozhraní a převádí jej na úroveň transakce. Zapouzdřuje hodnoty v objektu transakce. Provádí operace na úrovni transakce. Předává transakce do zobrazovače průběhů signálů. Odesílá transakce odběratelům prostřednictvím analyzis_port
Kolektor pokrytí	Je odběratelem transakcí z monitoru a vybírá vzorky pozorovaných transakcí a aktivit do

[4] Scoreboard - ověřovací komponenta, která obsahuje kontrolní nástroje a ověřuje funkčnost návrhu [9].

	skupin funkčního pokrytí. Sbírané údaje se ukládají do sdílené databáze pokrytí a používají se k určení celkového pokroku verifikace
Metrický analyzátor	Sleduje a zaznamenává nefunkční chování jako časování a využití energie. Provádí výpočty během fáze běhu anebo ve fázích po spuštění

Tab. 3: Části aktivního agenta UVM

Název	**Popis**
Driver	Je zodpovědný za komunikaci se sequncerem na úrovni transakce a za převod transakcí na úroveň signálů pro komunikaci s DUT, prostřednictvím virtuálního rozhraní. V případě potřeby může fungovat jako komponenta spojující návrh a tzv. responder[5]. V tomto případě driver reaguje na změny signálů na vstupních pinech, komunikuje se sekvencí responderu, který posílá transakce zpět do driveru. Potom driver posílá tuto odpověď DUT na úrovni signálů
Sequencer	Přepíná se mezi sekvencemi, které žádají o odeslání transakcí. Provádí obousměrnou komunikaci s driverem prostřednictvím TLM komunikace

UVMF generuje celý testbench podle popisu v YAML a propojuje jednotlivé části systému mezi sebou. Návrhář potom doplňuje potřebné části systému pro dosažení potřebné funkčnosti. Návrhář je schopen použít konfigurační proměnné, které byly definovány při popisu v YAML pro celé prostředí, sub-prostředí nebo agenta. Hodnoty konfiguračních proměnných se nastavují z testů a automaticky se předávají do konfiguračních objektů prostředí anebo agentů. Mechanismus konfigurace v UVMF působí vyšší úroveň konfigurovatelnosti celého systému. V případě, když cílem je implementovat vlastní konfigurační třídu s proměnnými a metodami, popis v YAML nestačí. V tomto případě návrhář vytvoří vlastní soubor, implementuje třídu a zahrne soubor do balíků částí systému, kde bude tuto konfigurační třídu používat. Tento konfigurační objekt vytvoří v testu, vyplní potřebné proměnné konfiguračního objektu a předá ho do částí systému, kde ho potřebuje používat. Pro předání konfiguračních

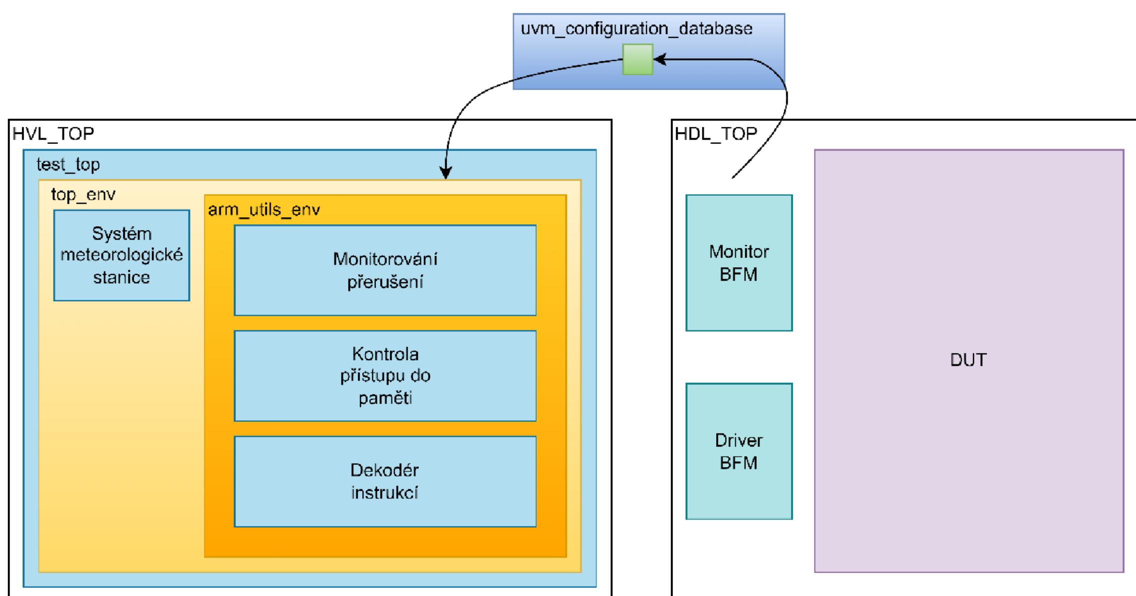
[5] Responder – ověřovácí komponenta, která interaguje se signály na rozhráni agenta, podobně jako driver [10].

objektů v UVM se používá konfigurační databáze. Jednoduchá konfigurovatelnost a parametrizovatelnost systému působí na vyšší úrovni opakovaného využití.

3.2 Popis sestaveného prostředí

Architekturou testbenche projektu byla vybrána architektura DualTop se dvěma moduly nejvyšší úrovně: HDL_TOP a HVL_TOP. HDL_TOP obsahuje modely a všechno, co přímo souvisí s činností RTL DUT. V části HDL_TOP jsou táky rozhrání BFM jednotlivých agentů. Modul HVL_TOP obsahuje objektově orientovaný testbench. Použití této architektury testbenche má následující výhody: nezávislost práce týmů verifikačních inženýrů a návrhářů číslicových obvodů, zmenšení času hledání chyby, možnost opakovaného využití verifikačních jednotek bez provádění velkých změn. Dvě vrcholové jednotky musejí být propojené mezi sebou pro HDL - HVL komunikaci. Podle metodologie UVM se to dělá pomocí konfigurační databáze. Virtuální rozhraní se přidávají do konfigurační databáze v části HDL a vyzvedávají se v HVL části.

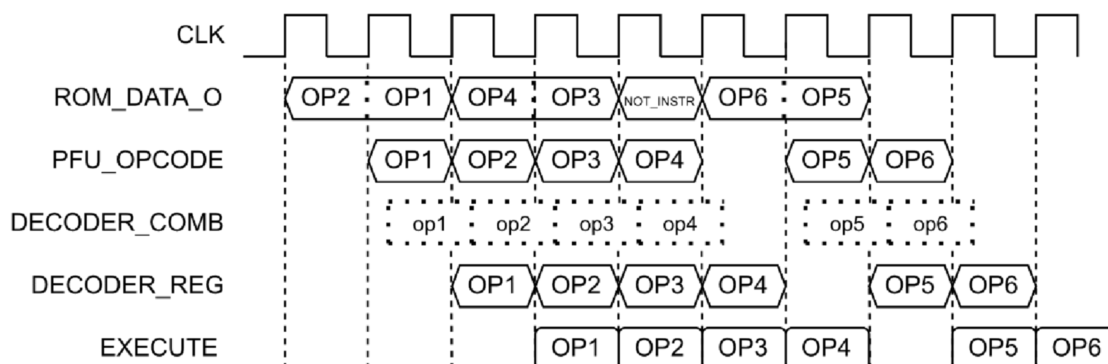
Na obrázku 6 je vidět realizované prostředí, které se zachovává hierarchické uspořádání HDL_TOP->Test->Prostredi->Agenti podle UVM. V rámci této práce byly navrhnuté a realizované 3 verifikační komponenty pro vizualizace a kontrolu činností procesoru: dekodér instrukcí, vizualizace přerušení, kontrola přístupu do paměti. Také byl sestaven model systému meteorologické stanice, využívající syntetizovatelné periferie procesoru SPI rozhraní a externí přerušení.



Obr. 6: Blokové schéma sestaveného prostředí

3.3 Dekodér instrukcí

Komponenta je schopna dekódovat 16bitový opkód a převádět ho do podoby assembrer instrukcí z instrukční sady Thumb a Thumb-2. Výstupem komponenty je dekódovaná instrukce a další závislé proměnné v podobě transakčních proměnných. Výstupní transakční proměnné jsou zabalené do transakčního objektu, který pomocí mechanismů UVMF a vybraného digitálního simulátoru lze zobrazit v grafickém rozhraní simulátoru.



Obr. 7: Časový diagram zpracování instrukcí

Cortex M0 má 3stupňovou pipeline. Zpracování a provedení instrukce probíhá ve třech fázích: načítání, dekódování a vykonání. Procesor vyčítá 32bitovou hodnotu z paměti programu ROM. Tou hodnotou mohou být: dvě 16bitové instrukce, jedna 32bitová instrukce, nebo data, kterými nejsou instrukce. Daty, kterými nejsou instrukce, mohou být: konstanta, identifikátor, počáteční hodnota proměnné. Posloupnost zpracování instrukcí je zobrazena na obrázku 7. Tato komponenta zobrazuje transakci s instrukcí ve fázi načítání. Výsledek vykonání instrukce je vidět po fázích dekódování a vykonání. Výsledkem může být změna hodnot systémových registrů, skok hodnoty čítače instrukce, změna hodnot stavových registrů APSR, volání systémových přerušení.

Položky výstupní transakce jsou popsané v tabulce 4. Hodnoty registrů v transakci odpovídají hodnotě registrů v okamžiku vizualizace, tj. v okamžiku načítání instrukce.

Tab. 4: Položky výstupní transakce dekodéru instrukcí

Datový typ	Název	Popis
string	cmd	Celá instrukce ve formátu Assembler
string	cmd_front	Instrukce ve formátu Assembler rozšířena o doplňující informace
structure	instr_regs	Struktura s vlastností instrukce
string	instr	Název instrukce
string	description	Krátký popis instrukce
string	Rd	Cílový registr*

string	Rt	Základní registr*
string	Rn	Registr, který obsahuje první operand*
string	Rm	Registr, který obsahuje druhý operand*
string	constant	Obsahuje hodnotu, kterou lze použít při aritmetických operacích nebo jako offset pro výpočet adresy*
structure	APSR	Stavový registr aplikačního programu
bit	N	Záporný příznak
bit	Z	Příznak nuly
bit	C	Příznak přenosu
bit	V	Příznak přetečení
structure	RegBank	Struktura obsahující systémové registry
bit [31:0]	reg_r00 - reg_r12	Registry pro obecné účely
bit [31:0]	reg_lr	Registr odkazů
bit [31:0]	reg_msp	Ukazatel na hlavní zásobník
bit [31:0]	reg_psp	Ukazatel zásobníku procesu
bit [31:0]	reg_aux	Pomocný registr
bit [31:0]	reg_pc	Programový čítač

* vyplňují se pouze ty položky, které byly použity k provedení instrukce

Položka transakce cmd_front se používá jako argument systémové funkce „$begin_transaction“, to znamená, že obsah této proměnné se používá jako název celé transakce a zobrazuje se jako první informace v grafickém rozhraní simulátoru. Příkladem instrukce ve formátu položky cmd je: „ADDS <Rd> <Rn> <Rm>“, stejná instrukce v rozšířeném formátu bude uchována v položce cmd_front: „ADDS r4 r0“. Vzhled transakcí dekodéru instrukcí v simulatoru Questa je uveden na obrázku 8.

Obr. 8: Vzhled transakcí dekodéru instrukcí v simulatoru Questa

Pro dekódování a zobrazení prováděných instrukcí ve správný okamžik komponenta využívá signály a registry uvnitř jádra. Seznam těchto registrů a signálů je popsán v tabulkách 5 a 6.

Tab. 5: Signály a registry, kterých využívá dekodér instrukcí

Datový typ	Název	Popis
bit [15:0]	opcode	Opkód instrukce
bit [15:0]	next_opcode	Dolní část 32bitového opkódu z ROM; další 16bitový opkód nebo část 32bitového opkódu v případě, že opkód[15:12] == 4'b1111
bit [30:0]	iaex	Adresa prováděné instrukce
bit [31:0]	r0 – r12	32bitové registry pro obecné účely
bit [31:0]	reg_lr	Registr odkazů
bit [31:0]	reg_ msp	Ukazatel na hlavní zásobník
bit [31:0]	reg_ psp	Ukazatel zásobníku procesu
bit [31:0]	aux_reg	Pomocný registr
bit [3 :0]	APSR	Stavový registr aplikačního programu (N, Z, C, V)

Komponenta je pasivní, to znamená, že neovlivňuje vstupy/výstupy DUT. Práce se signály, dekódování instrukcí a posílání transakcí na výstup je uskutečněna v monitoru. Funkčnost dekódovací části komponenty je zabalena do třídy v samostatném souboru. Tato třída obsahuje proměnné a metody pro dekódování. Hlavní dekódovací metoda volá odpovídající metody na základě prvních 3 až 12 bitů opkódu. Vhodná metoda dekóduje opkód a vyplní výstupní položky transakce. Dekodér vyplní pouze ty položky, které se podílejí na konkrétní instrukci. U instrukcí, kde se pracuje s adresou, je výstup uveden ve formátu: název instrukce, základní adresa nebo čítač instrukcí + vypočtený offset; konečná adresa. Příkladem takové instrukce je: B EQ {PC} + 0x4 ; 0x14A.

Hodnota opkódu bude dekódována komponentou a převedena na výstup v případě, že jsou všechny podmínky z tabulky 6 splněné, tzn., jednotlivé signály se nachází v potřebných stavech.

Tab. 6: Podmínky pro vizualizace transakce dekodérem instrukcí

Název signálu	Požadovaná hodnota	Význam
ctl_ex_last	1	Poslední perioda clk, kdy se dekóduje instrukce
pfu_op_special	0	Opkód není speciální
data_abort	0	Nenastala chyba na sběrnici během poslední periody dekódování instrukce
tbit	1	Jádro se nachází ve stavu Thumb, další stavy nejsou povolené v architektuře ARMv6M
int_preempt,	0‖1	Musí platit: jádro v daný okamžik se nepřipravuje na

atomic		obsloužení přerušení, nebo jádro se nachází v režimu atomic (používá se ve vícevláknových aplikacích)

3.3.1 Rozdělení instrukcí na typy a možnost změny barev vizualizace

Instrukce se dá rozdělit do 6 základních typů. Tyto typy jsou: instrukce větvení, instrukce zpracování dat, instrukce s možností přístupu do stavových registrů, instrukce načítání a ukládání, instrukce vícenásobného načítání a vícenásobného ukládání, další instrukce. Simulator Questa dovoluje měnit barvu vizualizovaných transakcí. Barvu transakce je možno nastavit uvnitř transakční třídy, uvnitř metody UVMF „add_to_wave“ pomocí systémové funkce „$add_color“. V rámci této komponenty barva transakce se nastaví v závislosti na názvu instrukce. Výsledná transakce může mít jednu z osmi definovaných barev: 6 barev pro 6 základních typů, jedna barva pro neznámé instrukce a jedna barva pro instrukci bod přerušení (BKPT). Uživatel je schopen lehce změnit libovolnou barvu pomocí změny hodnot lokálních parametrů uvnitř třídy.

3.3.2 Funkce výběru instrukcí pro vizualizace

Uživatel má možnost vybrat, které instrukce budou vizualizované v grafickém rozhraní simulátoru. K tomu je potřebné zavolat metodu „env_top_cfg_h.f_set_instr_dec_filter(string instr_name)“ uvnitř metody „build_phase“ hlavního testu a uvést název instrukce jako argument metody. Když metoda nebude zavolána ani jednou – všechny instrukce se budou vizualizovat. Kvůli tomu, že metoda se nachází uvnitř konfigurační třídy, je potřeba vytvořit objekt této třídy na začátku metody „build_phase“ pomocí systémové funkce new().

Názvy instrukcí, přidané pomocí této funkce, budou zapsané do fronty s položkami typu string. Tato fronta bude předána do konfigurační databáze a bude vyzvednuta na úrovni prostředí s těmito komponentami, které se nazývá arm_utils. Potom bude fronta předána do monitoru dekodéru instrukcí během propojovací fáze. Uvnitř funkce „notify_transaction“, která je umístěna v monitoru, je implementováno porovnání položky transakce, která obsahuje název instrukce a všech položek fronty. V případě, že název instrukce odpovídá jedné z položek fronty – transakce bude vizualizovaná, v opačném případě se transakce nezahodí, ale nebude vizualizována. Možnost výběru instrukcí pro vizualizaci usnadňuje hledání transakcí s potřebnou instrukcí v grafickém rozhraní simulátoru.

3.3.3 Funkce výpisu instrukcí v textovém formátu

Komponenta umožňuje zapnout vypisování prováděných instrukcí do textového okna simulátoru, nebo do terminálu v případě spuštění simulace bez grafického rozhraní. V případě používání funkce výběru instrukce, která byla popsána v kapitole 3.3.2 a zapnutého textového výstupu komponenty, pouze vybrané instrukce se budou vypisovat. Vypisovat se bude položka transakce cmd_front, což je Assembler instrukce

v rozšířeném formátu. Funkce se zapíná nastavením konfiguračního bitu v hlavním testu takto: „configuration.arm_utils_config.instr_dec_print_to_log = 1“.

3.3.4 Funkce časového omezení výpisu instrukcí

Když uživatel nastaví konfigurační bit pro výpis instrukcí v textovém formátu, tak má možnost nastavit časové intervaly, ve kterých se instrukce budou vypisovat. Pro nastavení tohoto intervalu je potřeba zavolat metodu „f_set_intr_dec_time_range“ konfigurační třídy uvnitř metody „build_phase“ hlavního testu. Metoda má dva vstupní argumenty typu realtime, tj. čas začátku a čas konce intervalu. Dalšími voláními metody je uživatel schopen nastavit požadovaný počet časových úseků, kdy se instrukce budou vypisovat.

3.4 Komponenta monitorování přerušení

Komponenta je schopná rozeznat vznik přerušení a poskytnout informaci o tomto přerušení uživateli ve formátu transakce. Komponenta monitoruje signály uvnitř procesoru a obsah registrů části NVIC. Na základě těchto informací skládá výslednou transakci.
Pro vizualizace transakcí ve správný okamžik a poskytnutí informace o přerušení komponenta využívá signály, které jsou popsané v tabulce 7.

Tab. 7: Vstupní signály komponenty monitorování přerušení

Datový typ	Název	Popis
bit [36:0]	int_active_bit	One-hot seznam aktivních výjimek: NMI, HDF, SVCall, PendSV, SysTick, IRQ[31:0]
bit	int_pend	Když je aktivní – NVIC se připravuje na obsluhu přerušení
bit [63:0]	int_irq_lvl	Jednotlivé úrovně priority IRQ
bit [1:0]	int_sys_tick_lvl	Úroveň priority přerušení od SysTick
bit [1:0]	int_svc_lvl	Úroveň priority přerušení SVCall
bit [1:0]	int_psv_lvl	Úroveň priority přerušení PendSV
bit	int_taken	Začátek obsluhy přerušení
bit	int_return	Konec obsluhy přerušení

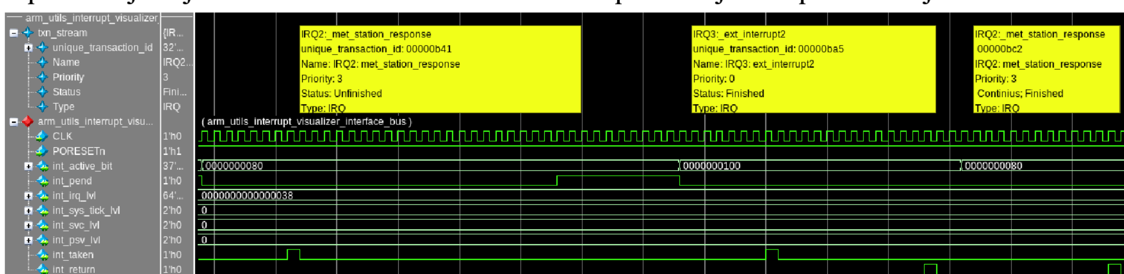
Komponenta je pasivní, práce se signály, vyplnění položek transakce a posílání transakcí na výstup jsou uskutečněny v monitoru. Začátek a konec vizualizace jsou určeny signály int_taken a int_return. Na základě dalších signálů z tabulky 7 se vyplňují položky transakce. Položky transakce jsou popsané v tabulce 8.

Tab. 8: Položky výstupní transakce komponenty monitorování přerušení

Datový typ	Název	Popis
string	Name	Název výjimky nebo název zdroje přerušení
string	Priority	Priorita přerušení

string	Status	Status přerušení (dokončené/nedokončené; -/pokračující)
string	Type	Typ přerušení: NMI, HDF, SVCall, PendSV, SysTick, IRQ

Na základě signálů int_irq_lvl, int_sys_tick_lvl, int_svs_lvl, int_psv_lvl je vyplněná položka Priority. Název a typ výjimky se zjišťuje na základě signálu int_active_bit. Názvem v případě přerušení typu NMI nebo IRQ může být název periferie, která toto přerušení vyvolala, jestli zdroj tohoto přerušení byl definován uživatelem v hlavním testu pomocí speciální metody. Komponenta je schopná rozeznat jestli přerušení bylo nedokončeno, tím pádem obsahem položky Status bude „unfinished“, tzn. Nedokončené. V opačném případě bude Status „finished“, což znamená dokončené. Při pokračování obsluhy nedokončeného přerušení, kromě statusu dokončené/nedokončené bude položka Status doplněna informací o tom, že obsluha přerušení byla nedokončená a pokračuje v jeho obsluze. Příklad vizualizace pokračujícího přerušení je na obrázku 9.



Obr. 9: Vizualizace transakcí komponenty monitorování přerušení

3.4.1 Možnost nastavení názvu zdroje přerušení

Komponenta umožňuje nastavit název zdroje přerušení NMI nebo IRQ.

Nastavení je třeba provést v hlavním testu uvnitř metody „build_phase“ pomocí metody „env_top_cfg_h.f_config_periph_nmi(string name_of_source)“ pro NMI a pomocí metody „env_top_cfg_h.f_config_periph_irq(string name_of_source, int irq_num)“ pro IRQ. Prvním argumentem metody je samotný název zdroje přerušení a v případě metody pro IRQ, druhým argumentem je číslo portu IRQ, ke kterému je zdroj přerušení připojen. Pro využití těchto metod je nutno před tím zavolat metodu „env_top_cfg_h.f_config_periph_create(string name_of_source, bit is_virtual)“. První argument musí být stejný pro všechny metody, protože se používá jako iterator asociativního pole konfiguračního objektu. Druhý argument musí mít hodnotu 0 v případě, že zdrojem přerušení je periferie, která má vyhrazený paměťový prostor v části paměti procesoru pro periferie. Druhý argument musí mít hodnotu 1, když je potřeba definovat systém vyšší úrovně abstrakce, který nemá přímo vyhrazený paměťový prostor.

Jestli uživatel definuje názvy zdrojů přerušení, položka transakce Name bude vyplněna odpovídajícím názvem a tento název bude použit jako argument systémové

funkce „$begin_transaction“, to znamená, že obsah této proměnné bude použit jako název celé transakce a bude se zobrazovat jako první informace v grafickém rozhraní simulátoru.

Stejně jako u komponenty dekodéru instrukcí je možnost nastavit barvy transakcí pomocí změny hodnot lokálních parametrů uvnitř třídy transakce „interrupt_visualizer_interface_transaction“. Lokální parametry pro každý typ přerušení je vidět na obrázku 10. Barvy je možné nastavit v hexidecimálním formátu RGB nebo uvést jednu z řady podporovaných barev simulátorem Questa.

```
localparam string color_hard_fault  = "#FF0000"; // red
localparam string color_main        = "#C2C2C2"; // grey
localparam string color_sv_call     = "#00A202"; // green
localparam string color_pend_sv     = "#FFAB19"; // orange
localparam string color_irq         = "#F1FF19"; // yellow
localparam string color_nmi         = "#F380FF"; // pink
localparam string color_sys_tick    = "#D3FC03"; // green+yellow
localparam string color_unexpect    = "#FF0000"; // red
```

Obr. 10: Lokální parametry pro nastavení barev transakcí

3.5 Komponenta kontroly přístupu do paměti

Úlohou této komponenty je monitorování adresy na sběrnici AMBA 3 AHB-Lite. V případě výskytu neočekávané adresy komponenta vyhlásí varování a zobrazí chybovou transakci v grafickém rozhraní simulátoru. Cílem je odhalit chyby při definici adresních rozsahů periferií procesoru.

Návrhář definuje adresní rozsahy periferií na straně programu a v části RTL v modulu AHB-Lite, který je zodpovědný za řízení signálů mezi sběrnicí a periferiemi. Blokové schéma systému je na obrázku 14, schéma AHB-Lite je na obrázku 2. Při špatném definování adresních rozsahů procesor může vyčítat nebo nahrávat data do špatných adres. ARM definuje, jak paměťový prostor procesoru musí být rozdělený. Důležité pro účely komponenty adresní hranice: ROM od 0x00000000 do 0x1FFFFFFF, SRAM od 0x20000000 do 0x3FFFFFFF, periferie od 0x40000000 do 0x5FFFFFFF, systém a privátní periferní sběrnice od 0xE000000 do 0xFFFFFFFF. Adresní rozsahy ROM a SRAM paměti jsou definovány návrhářem podle požadované velikosti paměti. Paměťový rozsah periferie je definován návrhářem podle počtu a velikosti stavových a konfiguračních registrů.

Vstupním signálem komponenty je 32bitový signál adresy na sběrnici HADDR. Výstupní položky transakce jsou uvedeny v tabulce 9.

Tab. 9: Položky výstupní transakce komponenty kontroly přístupu do paměti

Datový typ	Název	Popis
string	Source	Název periferie, které odpovídá aktuální adresa na sběrnici
string	State	Výsledek kontroly adresy: OK nebo mimo rozsah adres

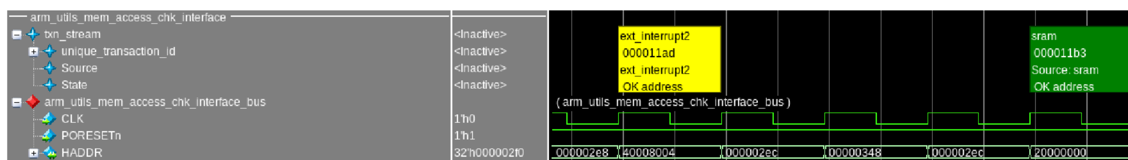
Pro činnost komponenty je nutné, aby uživatel definoval adresní rozsahy každé periferie včetně ROM a SRAM uvnitř metody „build_phase“ v hlavním testu. Uživatel musí zavolat metodu „env_top_cfg_h.f_config_periph_addr_range(string peripheral_name,logic[31:0] addr_bottom, logic[31:0] addr_top)“. Prvním argumentem metody je název periferie, druhým argumentem je dolní hranice adresního rozsahu, třetím argumentem je horní hranice adresního rozsahu periferie. Stejně jako v případě dalších komponent popsaných v této práci musí být nejdříve zavolána metoda „env_top_cfg_h.f_config_periph_create“ a argument, který obsahuje název periferie, musí být stejný pro všechny konfigurační metody, které se týkají dané periferie.

Adresní rozsahy se ukládají do konfigurační třídy prostředí, která se potom ukládá do konfigurační databáze. Komponenta porovnává adresu na sběrnici s rozsahy periferií z konfigurační databáze a hlásí varování v případě, že aktuální adresa není v žádném rozsahu adres. Varování nebude hlášeno v případě, že aktuální adresa se nachází v paměťovém prostoru systému, tj. od 0xE000000 do 0xFFFFFFFF.

3.5.1 Funkce zvýraznění komunikace procesoru s periferií

Komponenta umožňuje uživateli zobrazit komunikaci procesoru a vybrané periferie. Uživatel může zavolat metodu „env_top_cfg_h .f_set_mem_access_chk_filter(string peripheral_name, string color)“ uvnitř metody „build_phase“ v hlavním testu. Prvním argumentem je název periferie, druhým argumentem je barva transakce. Komponenta bude vizualizovat transakci v případě, že adresa na sběrnici se nachází uvnitř adresního rozsahu zadané periferie. Když tato metoda nebude zavolána ani jednou, budou se zobrazovat pouze chybové transakce. U této komponenty je možnost nastavit barvu transakce přímo z hlavního testu v hexidecimálním formátu RGB nebo uvést jednu z řady podporovaných barev simulátorem Questa. Přiklad této funkce je na obrázku 11.

Funkce zvýraznění komunikace procesoru s periferií umožňuje uživateli rychleji najít časový úsek, kde komunikuje s vybranou periferií, což způsobí rychlejšímu ladění.



Obr. 11: Funkce zvýraznění komunikace procesoru s periferií v Questa

3.6 Konfigurace prostředí arm_utils

Konfigurace prostředí arm_utils, které zahrnuje tři komponenty, je uskutečněna pomocí dvou vytvořených tříd. První třída se nazývá „periph_config“ a je určená k uchovávání konfigurační informace periferie. Třída je odvozena ze standardní třídy UVM – „uvm_object“. Třída obsahuje proměnné popsané v tabulce 10.

Tab. 10: Proměnné třídy periph_config

Datový typ	Název	Popis
string	periph_name	Název periferie
int	irq_num	Číslo přerušení IRQ, které používá periferie
bit	nmi	Nmi – 1, jestli periferie používá přerušení NMI; nmi – 0, jestli periferie nepoužívá přerušení NMI
logic [31:0]	addr_bottom	Dolní hranice adresního rozsahu periferie
logic [31:0]	addr_top	Horní hranice adresního rozsahu periferie
bit	virtual_periph	Virtual_periph - 0 v případě, že zdrojem přerušení je periferie, která má vyhrazený paměťový prostor v části paměti procesoru pro periferie. Virtual_periph - 1, když je potřeba definovat systém vyšší úrovně abstrakce, který nemá přímo vyhrazený paměťový prostor

Druhá třída se nazývá „env_top_config“, tato třída obsahuje proměnné a metody, které se používají ke konfiguraci verifikačních komponent. Uvnitř třídy jsou definované vlastní datové typy: „mem_chk_highlight_t“ a „instr_dec_filter_t“. Těmito datovými typy jsou fronty s položkami typu string. Dalšími vlastními datovými typy jsou „instr_dec_time_range_struct_t“ a „instr_dec_time_range_t“. Prvním datovým typem je struktura s položkami „start_time“ a „end_time“ typu realtime. Druhým datovým typem je fronta s položkami typu struktury „instr_dec_time_range_struct_t“. Proměnné třídy env_top_config jsou v tabulce 11.

Tab. 11: Proměnné třídy env_top_config

Datový typ	Název	Popis
periph_config [string]	cfg	Asociativní pole s položkami datového typu třídy „periph_config“ a iteratorem typu string
periph_config [int]	irq	Asociativní pole s položkami datového typu třídy „periph_config“ a iteratorem typu int
periph_config	nmi	Proměnná datového typu třídy „periph_config“
instr_dec_filter_t	instr_dec_filter	Proměnná typu fronta s položkami typu string
instr_dec_time_range_t	instr_dec_time_range	Proměnná typu fronta s položkami typu struktury „instr_dec_time_range_struct_t“

mem_chk_highlight_t	mem_chk_highlight	Proměnná typu fronta s položkami typu string
mem_chk_highlight_t	mem_chk_highlight_color	Proměnná typu fronta s položkami typu string

Třída implementuje metody popsané v tabulce 12. Prvním krokem při vytvoření konfigurace periferie je volání metody „f_config_periph_create". Dále budou popsané principy funkcí všech metod třídy.

f_config_periph_create - Prvním argumentem této metody je název periferie. Druhým argumentem je bit virtual_periph. Metoda vytvoří objekt třídy „periph_config", vyplní položky „periph_name" a „virtual_periph" tohoto objektu. Potom metoda uloží tento objekt do asociativního pole „cfg" na pozici, která je určena hodnotou názvu periferie, protože iterator asociativního pole je typu string.

f_config_periph_irq - Prvním argumentem je název periferie, druhým argumentem je číslo přerušení IRQ. Metoda zaprvé ověřuje, jestli periferie s názvem, který byl zadán jako první argument, existuje v asociativním poli „cfg". Když takový objekt neexistuje, bude volána informační hláška. Potom metoda uloží objekt z „cfg", který odpovídá zadanému názvu periferie, do asociativního pole „irq" na pozici „irq_num", což je druhým argumentem. Metoda vyplní položku „irq_num" pole „cfg" druhým argumentem metody.

f_config_periph_nmi - Metoda funguje skoro stejně jako „f_config_periph_irq" s tím rozdílem, že „nmi" není asociativní pole, ale proměnná typu „periph_config". Položka „nmi" pole „cfg" bude mít hodnotu 1.

f_get_irq_src_name - Metoda vyžaduje zadat číslo přerušení IRQ jako vstupní argument a vyhledá objekt odpovídající tomuto číslu v poli „irq". Výstupem metody bude string s číslem přerušení a názvem odpovídající periferie z položky „periph_name" pole „irq".

f_get_nmi_src_name	- Metoda vrátí název periferie z položky „periph_name“ proměnné „nmi“.
f_config_periph_addr_range	- Prvním argumentem metody je název periferie, druhým argumentem je dolní hranice adresního rozsahu, třetím argumentem je horní hranice adresního rozsahu periferie. Metoda ověří, jestli existuje objekt v poli „cfg“, který odpovídá zadanému názvu periferie. Když ne, tak zavolá informační hlášku. Potom metoda vyplní položky „addr_bottom“ a „addr_top“ objektu pole „cfg“ podle vstupních argumentů metody.
f_set_instr_dec_filter	- Vstupním argumentem je název instrukce typu string. Metoda přidá tento název do fronty „instr_dec_filter“.
f_set_mem_access_chk_filter	- Prvním argumentem je název periferie, druhým argumentem je barva transakce. Název bude přidán do fronty „mem_chk_highlight“, barva bude přidána do fronty „mem_chk_highlight_color“.
f_set_intr_dec_time_range	- Metoda slouží k přidání časového úseku, kdy je vypisování instrukcí povoleno uživatelem. Má dva vstupní argumenty typu realtime: „start_time“ a „end_time“. Metoda vytvoří a vyplní strukturu typu „instr_dec_time_range_struct_t“ a přidá tuto strukturu do fronty „instr_dec_time_range“.

Tab. 12: Metody konfigurační třídy env_top_config

Název metody	Argument 1	Argument 2	Argument 3	Popis
f_con-fig_periph_create	Název periferie (string)	Virtuální periferie (string)		Povinná metoda. Virtual periph = 1: vyšší úroveň abstrakce (nemá vlastní addr_range), Virtual periph = 0: běžná periferie
f_con-fig_periph_addr_range	Název periferie (string)	Dolní hranice adresního rozsahu (logic [31:0])	Horní hranice adresního rozsahu (logic [31:0])	Nastavení adresového prostoru všech periferií včetně paměti ROM a RAM

f_con-fig_periph_irq	Název periferie (string)	Číslo IRQ (int)		Nastavení čísel IRQ periferií
f_con-fig_periph_nmi	Název periferie (string)			Nastavení názvu periferie, která používá NMI
f_set_in-str_dec_filter	Název periferie (string)			Zavoláním této metody použijete funkce výběru instrukcí pro vizualizaci, která byla popsána v kapitole 3.3.2. Pokud byla metoda zavolána alespoň jednou, zobrazí se pouze přidaná instrukce a vypíše se do terminálu (jestli instr_dec_print_to_log == 1)
f_set_intr_dec_time_range	Čas začátku (realtime)	Čas konce (realtime)		Zavoláním této metody použijete funkci časového omezení výpisu instrukcí, popsanou v kapitole 3.3.4
f_set_mem_ac-cess_chk_filter	Název pe-riferie (string)	Barva (string)		Zavoláním této metody použijete funkci zvýraznění popsanou v části 3.5.1

3.6.1 Možnost vypnutí komponent

Systém komponent dovoluje vypnout instanciování jednotlivých komponent nastavením konfiguračních bitů. Konfigurační bity musejí být nastavené uvnitř metody „build_phase" hlavního testu. Příklad nastavení konfiguračních bitů je na obrázku 12. Když byl konfigurační bit nastaven na nulu, komponenta se nevytvoří během fáze sestavování prostředí.

```
    configuration.arm_utils_config.instr_dec_ena          = 1;
    configuration.arm_utils_config.instr_dec_print_to_log = 0;
    configuration.arm_utils_config.int_vis_ena            = 1;
    configuration.arm_utils_config.mem_acc_chk_ena        = 0;
```

Obr. 12: Příklad nastavení konfiguračních bitů pro vypnutí komponent

3.7 Návod na integraci komponent do systému uživatele

1) Zkopírujte soubor prostředí „arm_utils" a soubory rozhraní z arm_utils_package/CortexM0_M0+/verification_ip do svého projektu.
2) Připojte rozhraní k signálům uvnitř jádra. Všechny cesty k signálům rozhraní jsou v souboru if_signals.txt.

3) Zkopírujte soubory YAML z adresáře CortexM0_M0+/yaml do adresáře yaml vašeho prostředí.
4) Přidejte dílčí prostředí „arm_utils“ do souboru s popisem yaml vašeho hlavního prostředí.
5) Zkopírujte části „active_passive“, „interface params“ a „imports“ do popisu yaml vašeho testbenche z souboru CortexM0_M0+/support/bench_yaml.txt.
6) Spusťte skript yaml2uvmf.py.
7) Doplňte signálové spoje rozhraní v „hdl_top“ vašeho projektu pomocí souboru CortexM0_M0+/support/hdl_top_interfaces.txt. Je třeba zajistit přiřazení některých požadovaných signálů. Tato přiřazení jsou v hdl_top_interfaces.txt.
8) Přidejte několik rakdu kódu do project_benches/<váš_tbench>/tb/sequences/src/<váš_tbench>_bench_sequence _base.svh v metodě body() podle obrázku 13.
9) Vytvořte odkáz (handler) konfigurační třídy ve vašem hlavním testu takto: env_top_config env_top_cfg_h;.
10) Vytvořte konfigurační objekt na začátku metody „build_phase“ ve svém hlavním testu: env_top_cfg_h = new();
11) Volání konfiguračních metod komponent uvnitř „build_phase“ hlavního testu podle tabulky 15 po příkazu „super.build_phase(phase);“. Metody se nacházejí uvnitř konfigurační třídy, takže je volejte například takto: env_top_cfg_h. „nazev_metody“ („argumenty_metody“);. Metodu „f_config_periph_create“ je třeba zavolat před voláním jakékoli konfigurační metody pro danou periferií.
12) Nastavte konfigurační objekt v konfigurační DB uvnitř metody „build_phase“ těsně před voláním metody „configuration.initialize“ pomocí: uvm_config_db #(env_top_config)::set(this , „*“, „cm0_env_cfg“, env_top_cfg_h);.
13) Spusťte simulaci pomocí makefile, který byl vygenerován UVMF. Použijte argument „-define CM0“ nebo „-define CM0PLUS“ při sestavování projektu.

```
// Start INITIATOR sequences here
    fork
      if(arm_utils_decoder_interface_config.instr_dec_ena)
      begin
        repeat(25) arm_utils_decoder_interface_random_seq.start(
                   arm_utils_decoder_interface_sequencer);
      end
      if(arm_utils_interrupt_visualizer_interface_config.int_vis_ena)
      begin
        repeat(25)
          arm_utils_interrupt_visualizer_interface_random_seq.start(
          arm_utils_interrupt_visualizer_interface_sequencer);
      end
      if(arm_utils_mem_access_chk_interface_config.mem_acc_chk_ena)
      begin
        repeat(25)
```

```
        arm_utils_mem_access_chk_interface_random_seq.start(
        arm_utils_mem_access_chk_interface_sequencer);
    end
  join
```

Obr. 13: Úprava metody body() tridy „cm0_tbench_bench_sequence_base“

3.8 Simulace v Cadence Xcelium

UVMF je zaměřený na simulaci v Siemens Questa, proto nejsou vygenerované skripty na simulace v dalších simulátorech, třeba Xculium. Nad rámec zadaní byly komponenty přizpůsobené a byl napsan skript, který slouží k sestavení projektu pro simulaci v Cadence Xcelium. Skript byl realizován v jazyce TCL, který obsahuje potřebné proměnné, metody a další příkazy na generování spustitelného souboru „build.do“.

Pomocí dalších skriptů firmy ON Design Czech s.r.o. byly k dispozici tzv. soubory souborů (file of files). Skript v jazyce TCL popisuje procedury, které přidávají soubory s potřebnými argumenty do souboru „build.do“. Tento soubor je základem sestavování projektu, potom následují fáze elaborace a samotné simulace.

Pro simulace v Xcelium bylo zapotřebí změnit tvar výstupních transakcí komponent. Kvůli tomu, že Xcelium nepodporuje rozbalené struktury jako tvar výstupní transakce a datový typ string, struktury byly nahrazené za jednotlivé výstupy, tzn., bez hierarchického uspořádání. Datový typ string byl nahrazen vlastním datovým typem. Tímto datovým typem je pole symbolů stanoveného rozměru.

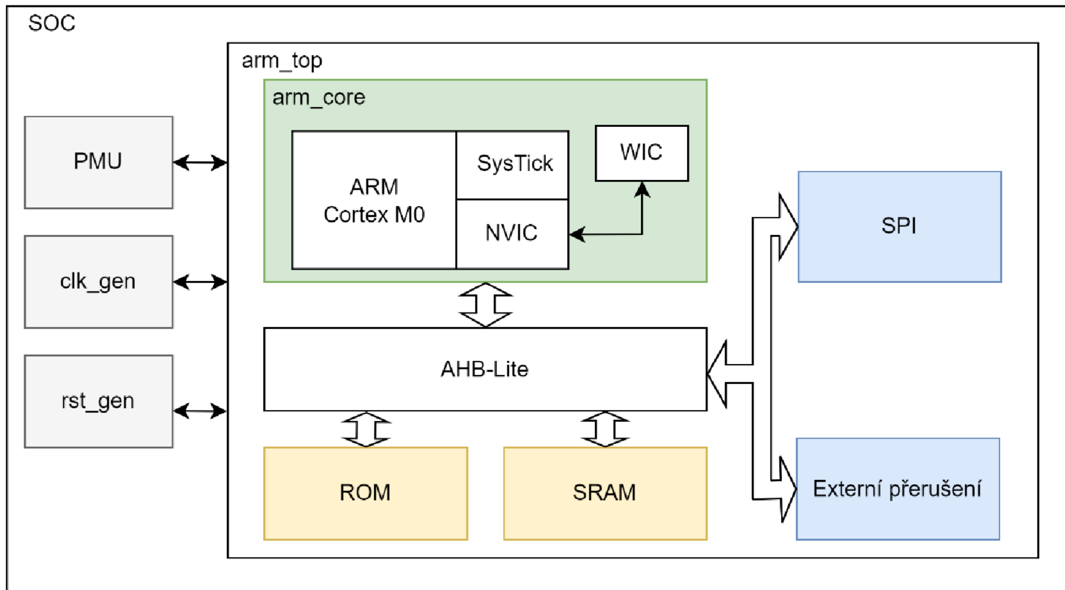
V přílohách jsou k dispozici implementované skripty v jazyce TCL pro Cortex-M0 a pro Cortex-M0+.

3.9 Referenční návrh

Pro testování a ukázku činností navržených verifikačních komponent byl sestaven referenční návrh.

Povinnými částmi návrhu s procesorem ARM jsou: paměť určená pouze pro čtení dat – ROM, statická paměť pro zápis a čtení uložených dat – SRAM, matrice komunikační sběrnice AMBA AHB-Lite, ovladač resetování (rst_gen), generátor hodinových signálů (clk_gen) a jednotka řízení spotřeby (PMU).

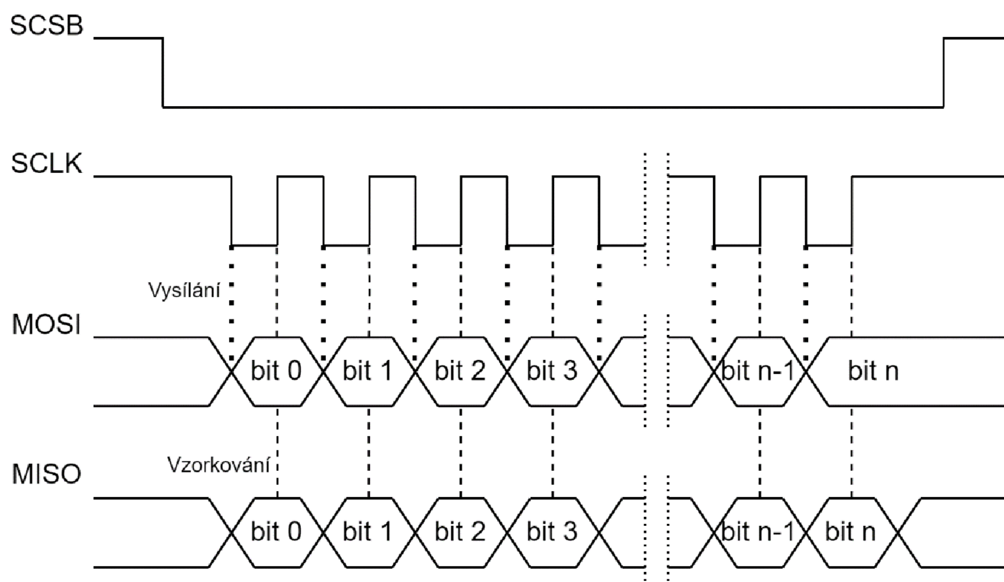
Do referenčního návrhu byly integrovány dvě vlastní periferie – SPI a Externí přerušení. Tyto periferie budou využívat přerušení IRQ, v podprogramech obsluhy přerušení těchto periferií bude uskutečněna práce s jejich registry a s proměnnými z paměti SRAM. Blokové schéma procesoru s navrženými periferiemi je na obrázku 14.



Obr. 14: Blokové schéma referenčního návrhu s procesorem

3.9.1 SPI rozhraní

SPI rozhraní v tomto návrhu představuje nadřízenou jednotku, to znamená, určuje časové parametry přenášených rámců, jejich šířku a parametry SCLK a SCSB signálů.
Periferie vysílá data z portu MOSI na sestupnou hranu signálu SCLK, vzorkuje data od podřízeného externího zařízení na vzestupnou hranu SCLK, jak je ukázáno na obrázku 15.



Obr. 15: Ukázka jednoho SPI rámce

Navržená periferie je určena pro komunikaci s externími zařízeními. Data se posílají z ROM nebo SRAM do procesoru, adresa a data se potom posílají přes sběrnici do periferie. Podle adresy se určuje, jestli jsou data konfigurační nebo na vysílání externímu zařízení.

Data na vysílání se ukládají do registrů vyrovnávací paměti typu „První dovnitř, první ven“ (FIFO). Hloubka FIFO paměti je parametrizovatelná, ale ve výchozím nastavení je 16, šířka paměti je konstantní a rovná se 32bit. To znamená, že maximální šířka rámce je omezena na 32 bitů. Po skončení rámce periferie zavolá přerušení, aby procesor vybral data z druhé vyrovnávací paměti, kam se ukládají vzorkované rámce.

Nastavení parametrů rámce a periferie je uskutečněno pomocí konfiguračních registrů z tabulky 15, monitorování stavu obsazenosti vyrovnávacích pamětí pomocí stavových registrů z tabulky 14. Registry pro vysílací a přijatá data SPI jsou v tabulce 13.

Tab. 13: 32bitové registry pro vysílací a přijatá data SPI

Bit	Název registru	Popis registru
[31:0]	SPI_WDATA	Slouží na posílání dat do SPI periferie
[31:0]	SPI_RDATA	Slouží na vyčítání dat z paměti SPI

Tab. 14: Stavový registr SPI_STAT_REG

Bit	Název registru	Popis registru
[7:0]	SPI_READ_FRM_CNT	Počet rámce v paměti pro přijatá data
[8]	SPI_FIFO_FULL	Paměť pro vysílaná data je plná, 1 – plná
[9]	SPI_FIFO_EMPTY	Paměť pro vysílaná data je prázdná, 1 - prázdná
[10]	SPI_READY	Poslední poslaný rámec se ukončil a přijatá data jsou připravená k odběru, 1 - připravený
[31:11]	nevyužité	Nevyužité registry

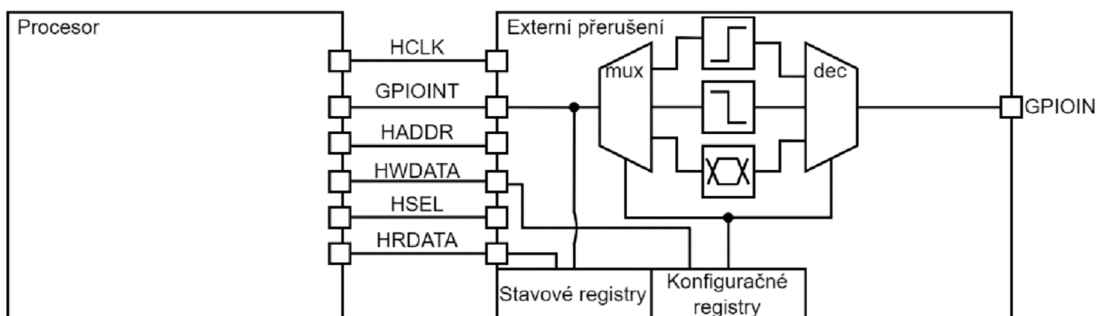
Tab. 15: Konfigurační registr SPI_CFG_REG

Bit	Název registru	Popis registru
[7:0]	SPI_FRAME_LEN	Nastavení šířky rámce
[8]	SPI_INT_ENA	Povolení generovat přerušení, 1 – povoleno
[31:9]	nevyužité	Nevyužité registry

3.9.2 Externí přerušení

Periferie je určena pro detekci aktivní úrovně nebo hrany signálu z externího vstupu procesoru a následném generování přerušení.

Za prvé musí být nastavená hrana, respektive úroveň, na kterou periferie bude citlivá a musí být povoleno přerušení. Návrh dovoluje vybrat maximálně dvě hrany a dva úrovně, na které bude periferie citlivá. Nastavení periferie je realizováno s využitím konfiguračních registrů, popsaných v tabulce 16. V podprogramu pro obsluhu přerušení je příkaz na načtení odpovídajícího stavového registru periferie. Stavové registry periferie jsou definované v tabulce 17. Koncepční schéma této periferie je na obrázku 16.



Obr. 16: Koncepční schéma periferie externího přerušení

Tab. 16: Konfigurační registr externího přerušení EXT_INT_CFG_REG

Bit	Název registru	Popis registru
[0]	EXT_INT_RE_SNS	Citlivost na nástupnou hranu, 1 – citlivé, 0 - necitlivé
[1]	EXT_INT_FE_SNS	Citlivost na sestupnou hranu, 1 – citlivé, 0 - necitlivé
[2]	EXT_INT_LOW_SNS	Citlivost na nízkou úroveň, 1 – citlivé, 0 - necitlivé
[3]	EXT_INT_HIGH_SNS	Citlivost na vysokou úroveň, 1 – citlivé, 0 - necitlivé
[4]	EXT_INT_ENA	Povolení generovat přerušení, 1 – povoleno
[31:5]	nevyužité	Nevyužité registry

Tab. 17: Stavový registr externího přerušení EXT_INT_STAT_REG

Bit	Název registru	Popis registru
[0]	EXT_INT_RE_ST	Stav detekce nástupné hrany, 1 – nastala, 0 - nenastala
[1]	EXT_INT_FE_ST	Stav detekce sestupné hrany, 1 – nastala, 0 - nenastala
[2]	EXT_INT_LOW_ST	Stav detekce na nízkou úroveň, 1 – nastala, 0 – nenastala
[3]	EXT_INT_HIGH_ST	Stav detekce na vysokou úroveň, 1 – nastala, 0 – nenastala
[31:4]	nevyužité	Nevyužité registry

3.9.3 Systém meteorologické stanice

V rámci této práce byl sestaven systém meteorologické stanice, který využívá periférii popsané v kapitolách 3.9.1 a 3.9.2. Systém obsahuje dvě části: první část je procesor a periferie, druhá část je agent „met_station“, který obsahuje tzv. responder. Agent se nachází v části HVL a modeluje senzory reálné meteorologické stanice a reciever/transmitter SPI linky. Procesor a periferie představují nadřízenou jednotku.

Systém používá registry SPI rozhraní a externího přerušení, popsané v tabulkách 11, 12, 13, 14, 15. Procesor posílá požadavek na načtení dat z meteorologické stanice přes linku SPI. Délka rámce SPI je 32 bitů. Rámce pro čtení a zápis jsou rozdělené podle tabulek 18 a 19.

Tab. 18: Rozdělení rámce pro zápis, meteorologická stanice

Bit	Název registru	Popis registru
[7:0]	MOSI_ADDR	Adresa, podle které bude vybrán meteorologický prvek na vyčtení
[8]	RW	Když tento bit má hodnotu 1 – změna hodnot selektoru sel_ra a sel_rb je povolena
[11:9]	SEL_RB	Selektor registru RB
[14:12]	SEL_RA	Selektor registru RA
[31:15]	nevyužité	Nevyužité registry

Tab. 19: Rozdělení rámce pro čtení, meteorologická stanice

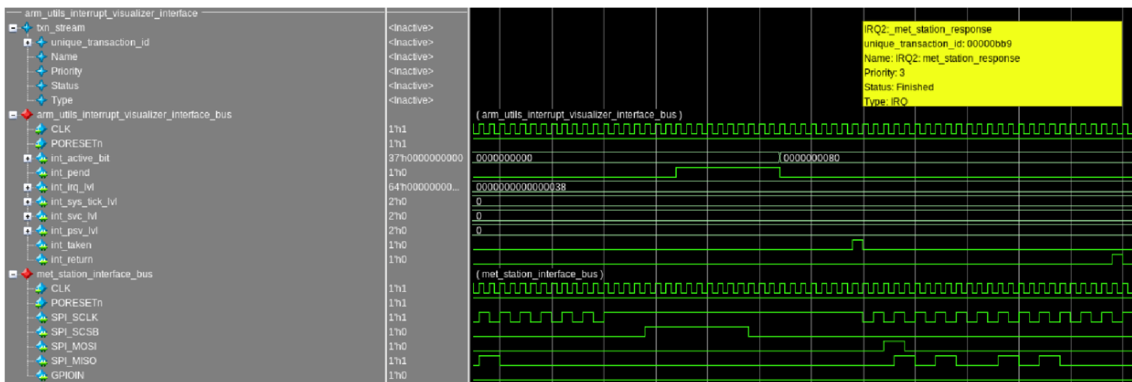
Bit	Název registru	Popis registru
[7:0]	CRC	Cyklický redundantní součet
[15:8]	MISO_ADDR	Adresa meteorologického prvku, který byl poslán v tomto rámci
[31:16]	DATA	Data vyčtená z meteorologické stanice

Procesor posílá adresu, podle které meteorologická stanice rozhoduje, jaká data musí poslat v dalším rámci. Těmito daty mohou být prvky: teplota, vlhkost, tlak, rychlost větru, směr větru nebo dva 8bitové registry RA a RB. Teplota je představena ve formátu reálného čísla s pevnou řádovou čárkou: 8 bitů celá část, 8 bitů zlomková část. Další prvky jsou jenom ve formátu přirozených čísel. Obsahem registrů RA, RB může být celá část teploty, vlhkost, tlak, rychlost větru, směr větru podle hodnot selektoru registru SEL_RA a SEL_RB. To znamená, že procesor vybírá buď vyčíst přesnou hodnotu jednoho prvku během jednoho rámce, nebo vyčíst hodnoty dvou prvků s menší přesností. Na konci rámce je zavoláno přerušení. Podprogram, který odpovídá tomuto přerušení, obsahuje příkazy na vyčítání všech rámců z FIFO paměti periferie „SPI rozhrání“. Principy funkce periferie jsou popsané v kapitole 3.9.1. Součástí rámce od meteorologické stanice je vypočtený cyklický redundantní součet předávaných dat

s konstantním klíčem. Procesor dostane data od meteorologické stanice a může vypočítat cyklický redundantní součet. Porovnáním vypočtené a obdržené hodnoty CRC bude rozhodnuto o správnosti přijatých dat. V případě rozdílné hodnoty CRC může požádat o zaslání nových dat.

Uživatel je schopen nastavit periodu vyčítání parametrů z meteorologické stanice. Když nebude z programu poslán žádný rámec na vyčítání během nastavené doby, automaticky bude vygenerován impuls na jednom portu IRQ. Změnu signálu na tomto portu zaznamená periferie „Externí přerušení". Procesor obslouží toto přerušení, tzn., vykoná podprogram obsluhy přerušení. Obsahem podprogramu může být například vyčítání hodnot registrů RA, RB.

Na obrázku 17 je ukázána informace o zpracování přerušení, které je voláno na konci SPI rámce. Tuto informaci poskytuje komponenta monitorování přerušení. Na obrázku 17 je vidět, že obsluha přerušení začala po rámci SPI. Začátek a konec obsluhy jsou dané signály int_taken a int_return. Transakce komponenty monitorování přerušení obsahuje název zdroje přerušení, prioritu, typ přerušení a informaci o tom, že obsluha přerušení byla dokončena.

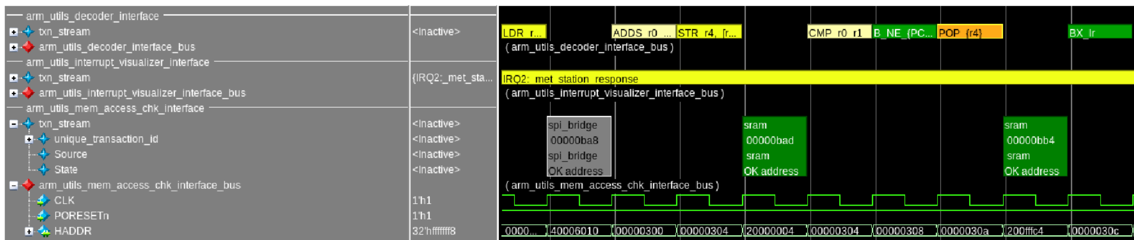


Obr. 17: Vizualizace obsluhy přerušení po rámci SPI, meteorologická stanice

Na obrázku 18 je ukázána činnost komponenty dekodéru instrukcí během obsluhy přerušení. Během obsluhy přerušení se vykonávají instrukce z podprogramu obsluhy konkrétního přerušení.

Obr. 18: Činnost komponenty dekodéru instrukcí během obsluhy přerušení, meteorologická stanice

Na obrázku 19 je vidět vizualizaci přístupu do pozorovaných oblastí paměti procesoru. Je vidět, že během obsluhy přerušení procesor přistupuje do oblasti paměti SRAM a do registru SPI rozhraní.



Obr. 19: Činnost komponenty kontroly přístupu do paměti během obsluhy přerušení, meteorologická stanice

4. ZÁVĚR

Tato práce byla zadána spolupracující firmou ON Design Czech s.r.o. Cílem práce bylo analyzovat možnosti verifikace systému s procesorem ARM Cortex-M0/M0+ a navrhnout konfigurovatelné komponenty pro debuggování a kontrolu činnosti.

Teoretická část práce popisuje funkce procesoru a vysvětluje principy činností jeho základních částí. Popis činností procesoru je pro lepší pochopení prokládán obrázky. Důraz byl kladen na popis komunikace procesoru s periferiemi a obsluhy přerušení, které mohou být generované periferiemi nebo bezprostředně uvnitř procesoru. Byly popsané veškeré možné druhy těchto výjimek a přerušení. V této části byly také popsány funkce pro nízkou spotřebu energie a speciální registry.

Součásti práce je dále vlastní návrh řešení systému. Ten definuje koncept verifikačních komponent, které je možné rozdělit do dvou skupin: komponenty kontrolující správnou činnost procesoru a komponenty vizualizační, které budou zobrazovat důležité informace během simulace a zlehčovat ladění. Kapitola se zabývá principem funkcí komponent a způsobem realizace návrhu.

V praktické části bakalářské práce bylo popsáno prostředí UVM, ve kterém byly verifikační komponenty realizované. Těmito komponenty jsou: dekodér instrukcí, komponenta monitorování přerušení a komponenta kontroly přístupu do paměti. Principy funkce, možnosti konfigurace, vstupy a výstupy komponent byly detailně popsány v této části. Konfigurace prostředí s těmito komponenty, která dovoluje uživateli nastavit komponenty podle svých požadavků, byla následovně vysvětlena. Navazující kapitolou je návod na integraci prostředí s komponentami do prostředí uživatele. Realizovaná práce byla optimalizována pro dva digitální simulátory: Siemens Questa a Cadence Xcelium. Způsob optimalizace pro Xcelium je popsán v této části práce. Praktická část je ukončena popisem sestaveného referenčního návrhu, který byl realizován pro testování a ukázku činnosti verifikačních komponent. Podrobně definuje navržené periferie SPI a externí přerušení, které byly realizované a integrované do systémů s procesorem ARM Cortex-M0. S využitím těchto periferií a agenta z části HVL byl sestaven systém meteorologické stanice. Potom je demonstrována činnost verifikačních komponent na tomto referenčním návrhu.

LITERATURA

[1] Cortex-M0 Devices Generic User Guide Documentation – Arm Developer. Arm Developer [online] [cit. 03.12.2022]. Dostupné z: https://developer.arm.com/documentation/dui0497/a/?lang=en

[2] YIU, J. (2015). The definitive guide to ARM (R) cortex (R)-M0 and cortex-M0+ processors (2nd ed.). Newnes. [cit. 08.12.2022].

[3] Architektura mikrořadičů s jádry ARM Cortex-M0 a ARM Cortex-M0+ - Root.cz. Root.cz - informace nejen ze světa Linuxu [online]. Copyright © 1997 [cit. 08.12.2022]. Dostupné z: https://www.root.cz/clanky/architektura-mikroradicu-s-jadry-arm-cortex-m0-a-arm-cortex-m0/#k05

[4] What is a Pipeline? - Definition from Techopedia. Techopedia: Educating IT Professionals To Make Smarter Decisions [online]. Copyright © 2022 [cit. 03.12.2022]. Dostupné z: https://www.techopedia.com/definition/5312/pipeline

[5] What's the difference between Von-Neumann and Harvard architectures?. Tips on coding, designing, and embedding with microcontrollers [online] [cit. 03.12.2022]. Dostupné z: https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/

[6] Von-Neumann vs Harvard Architecture | Differences & Uses. GCSE Computer Science Revision & Resources | Computer Science UK [online]. Copyright © 2022 Teach Computer Science [cit. 03.12.2022]. Dostupné z: https://teachcomputerscience.com/von-neumann-harvard-architecture/

[7] AMBA 3 AHB-Lite Protocol Specification Documentation – Arm Developer. Arm Developer [online] [cit. 03.12.2022]. Dostupné z: https://developer.arm.com/documentation/ihi0033/a/Introduction

[8] UVM | Verification Academy. Verification Academy - The most comprehensive resource for verification training. | Verification Academy [online]. Copyright © Siemens 2023 [cit. 27.05.2023]. Dostupné z: https://verificationacademy.com/cookbook/uvm

[9] UVM Scoreboard. ChipVerify [online]. Copyright © 2015 [cit. 28.05.2023]. Dostupné z: https://www.chipverify.com/uvm/uvm-scoreboard

[10] Glossary/Responder | Verification Academy. Verification Academy - The most comprehensive resource for verification training. | Verification Academy [online]. Copyright © Siemens 2023 [cit. 28.05.2023]. Dostupné z: https://verificationacademy.com/cookbook/doc/glossary/responder

[11] Phasing | Verification Academy. Verification Academy - The most comprehensive resource for verification training. | Verification Academy [online]. Copyright © Siemens 2023 [cit. 28.05.2023]. Dostupné z: https://verificationacademy.com/cookbook/phasing

SEZNAM SYMBOLŮ A ZKRATEK

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APSR	Application Program Status Register
ARM	Advanced RISC Machine
CRC	Cyclic Redundancy Check
DUT	Design Under Test
FIFO	First In First Out
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
HVL	Hardware Verification Language
IRQ	Interrupt Request
LR	Link Register
MPU	Memory protection unit
MSB	Most Significant Bit
NMI	Non-Maskable Interrupt
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PC	Program Counter
PMU	Power Management Unit
RAM	Random Access Memory
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register-Transfer Level
SCB	System Control Block
SCS	System Control Space
SoC	System on Chip
SP	Stack Pointer
SPI	Serial Peripheral Interface
TCL	Tool Command Language
UVM	Universal Verification Methodology
UVMF	Universal Verification Methodology Framework
WFE	Wait For Interrupt
WFI	Wait For Event
WIC	Wakeup Interrupt Controller
YAML	YAML Ain't Markup Language

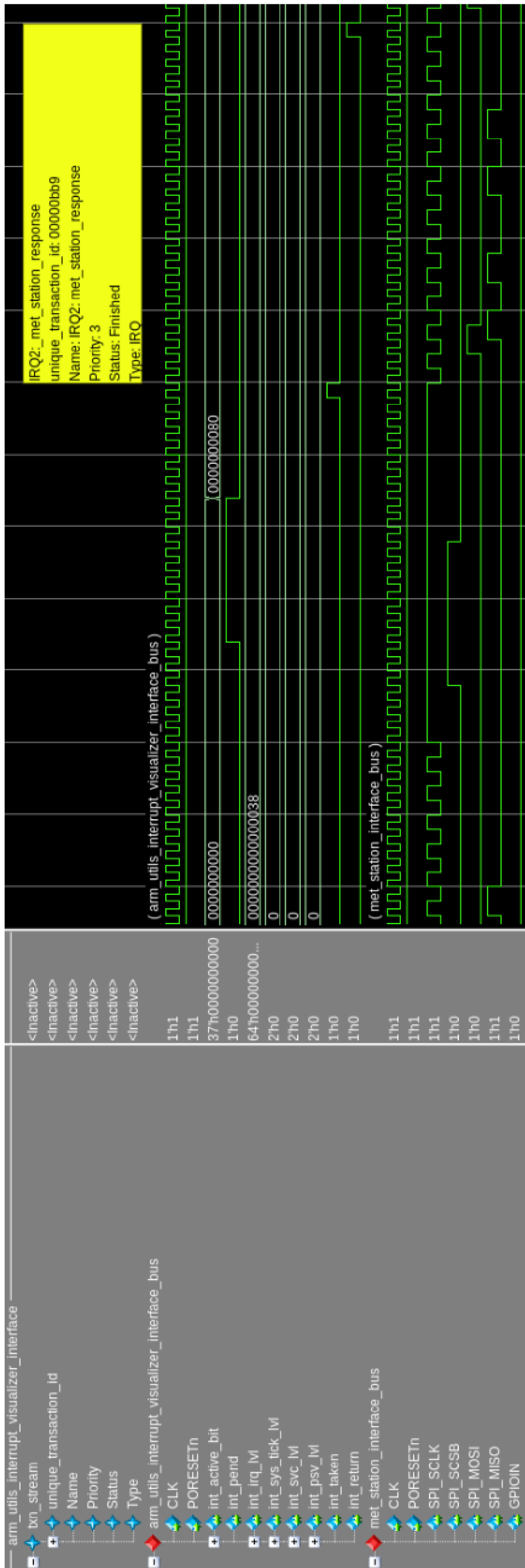
SEZNAM PŘÍLOH

PŘÍLOHA A - VÝSTUPY KOMPONENT, METEOROLOGICKÁ STANICE................................ 53

Příloha A - Výstupy komponent, meteorologická stanice

A.1 Činnost komponenty dekodéru instrukcí během obsluhy přerušení, meteorologická stanice

A.2 Vizualizace obsluhy přerušení po rámci SPI, meteorologická stanice



A.3 Činnost komponenty kontroly přístupu do paměti během obsluhy přerušení, meteorologická stanice