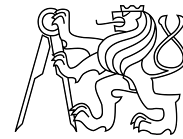
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Návrh generického digitálního SoC obvodu pro FPGA platformu
Student: Bc. Václav Vanc
Vedoucí: Ing. Martin Hujer, Ph.D.
Studijní program: Informatika
Studijní obor: Návrh a programování vestavných systém
Katedra: Katedra íslicového návrhu
Platnost zadání: Do konce letního semestru 2017/18

Pokyny pro vypracování

Navrhn te na úrovni RTL v jazyce Verilog2001 generický digitální parametrizovaný SoC (System on Chip) obvod, který bude umož ovat integraci procesorového jádra a r zných druh komunika ních rozhraní (jako nap íklad UART, I2C, SPI) na úrovni RTL. Systémová platforma pro integraci SoC obvodu bude napsaná v jazyce Python. Celý SoC obvod bude implementován do FPGA obvodu. Vybraný FPGA obvod a FPGA vývojová deska by m ly poskytovat velké množství digitálních vstupních a výstupních signál . SoC obvod v FPGA bude ovládán z p ipojeného po íta e p es USB rozhraní s využitím USB/UART p evodníku na desce FPGA. Prost edí pro ovládání z po íta e bude napsáno v jazyce Python.
Správnou funkcionalitu ešení ov te simulací na úrovni RTL a validací na FPGA desce.
V teoretické ásti práce m že být provedeno srovnání r zných architektur procesorových jednotek, vnit ních sb rnicových systém a sériových rozhraní, se kterými je možno se v SoC obvodech setkat.

Seznam odborné literatury

Dodá vedoucí práce.

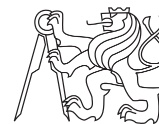
L.S.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 4. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu



Diplomová práce

Návrh generického digitálního SoC obvodu pro FPGA platformu

Bc. Václav Vanc

Vedoucí práce: Ing. Martin Hujer, Ph.D.

8. května 2017

Poděkování

Rád bych touto cestou poděkoval Ing. Martinovi Hujerovi, Ph.D. za vedení mé diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Václav Vanc. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Vanc, Václav. *Návrh generického digitálního SoC obvodu pro FPGA platformu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato diplomová práce se zabývá návrhem nástroje pro tvorbu systémů na čipu (SoC). Vygenerovaný SoC je možné ovládat prostřednictvím počítače pomocí rozhraní *UART*. Prostředí pro ovládání z počítače je napsáno v jazyce *Python*. Práce se zaměřuje jak na návrh hardwarové části SoC, tak na softwarovou část. Vytvořený SoC integruje procesor *ARM Cortex-M0* a periferie třetích stran. Nástroj pro generování SoC je napsaný v jazyce *Python* a je koncipován modulárně, což umožňuje přidávat podporu pro další periferie a sběrnice.

Klíčová slova SoC, systém na čipu, FPGA, ARM, Cortex-M, AMBA, AHB, APB, RTL, Python, UART, SPI, I^{2}C

Abstract

This diploma thesis deals with the development of a tool for creating systems on a chip (SoC). The generated SoC can be controlled by the computer via an *UART* interface. The program for control of the SoC is written in *Python*. The thesis deals with the design of both hardware and software part of the SoC. The created SoC integrates the *ARM Cortex-M0* processor and third-party peripherals. The SoC generator is written in *Python* and is designed to be modular. This allows easy implementation of support for additional peripherals and buses.

Keywords SoC, system on a chip, FPGA, ARM, Cortex-M, AMBA, AHB, APB, RTL, Python, UART, SPI, I^{2}C

Obsah

Úvod **1**

1 Analýza **3**
1.1 Generátory SoC . . . 3
1.2 Procesory . . . 4
1.3 Sběrnice . . . 7
1.4 Sériová rozhraní . . . 13
1.5 Vývojové FPGA desky . . . 17

2 Realizace **19**
2.1 Architektura SoC . . . 19
2.2 Komponenty . . . 20
2.3 Firmware . . . 29
2.4 Generátor SoC . . . 43
2.5 FPGA desky . . . 52
2.6 Ovládání z PC . . . 54

3 Verifikace a validace **57**
3.1 Verifikace . . . 57
3.2 Validace . . . 59

Závěr **65**

Literatura **67**

A Seznam použitých zkratek **71**

B Obsah přiloženého CD **73**

Seznam obrázků

1.1 Návrh v nástroji Vivado Design Suite IP Integrator 4
1.2 Ukázka SoC v nástroji Qsys 5
1.3 Kanály sběrnice AXI při čtení [8] 8
1.4 Kanály sběrnice AXI při zápisu [8] 9
1.5 AXI - Různé příklady potvrzování dat v kanálu 9
1.6 Sběrnice AHB [9] 10
1.7 Zápis na sběrnici AHB (bez a s prodlouženou datovou fází) [11] . . 10
1.8 Čtení na sběrnici AHB (bez a s prodlouženou datovou fází) [11] . . 10
1.9 Zápis na sběrnici APB [12] 11
1.10 Čtení na sběrnici APB [12] 11
1.11 Zápis na sběrnici APB se dvěma čekacími cykly [12] 12
1.12 Čtení na sběrnici APB se dvěma čekacími cykly [12] 12
1.13 Ukázka zapojení UARTu 13
1.14 Ukázka komunikace UARTu 13
1.15 Posuvné registry rozhraní SPI 14
1.16 Čtyři možné průběhy signálů SPI [16] 15
1.17 Ukázka zapojení rozhraní SPI 15
1.18 Ukázka zapojení rozhraní I^2C 16

2.1 Použití vytvořeného SoC pro validaci ASICu 19
2.2 Příklad vytvořeného SoC 20
2.3 Funkční bloky procesoru Cortex-M0 [18] 21
2.4 Doporučené rozložení paměti procesoru Cortex-M0 [19] 21
2.5 Časový diagram zápisu do AHB/APB mostu 24
2.6 Časový diagram čtení z AHB/APB mostu 24
2.7 Zapojení AHB s arbitry [10] 25
2.8 Ukázka systému s více mastery [10] 25
2.9 Ukázka arbitrace 26
2.10 Zápis do blokové paměti FPGA [20] 27
2.11 Čtení z blokové paměti FPGA [20] 27

2.12 Obálka (wrapper) GPIO 30
2.13 Proces vytváření SoC 44
2.14 Hierarchie sběrnic a mostů 45
2.15 Hierarchie s připojenými periferiemi GPIO, UART a SPI 45
2.16 Grafické rozhraní pro konfiguraci 47
2.17 Vývojová deska Nexys Video Artix-7 FPGA [29] 53
2.18 Vývojová deska DE1-SoC [30] 54

3.1 Ukázkový SoC pro Nexys Video Artix-7 FPGA 60
3.2 Nexys Video Artix-7 FPGA s připojeným UARTem 61
3.3 Rozložení SoC v FPGA na desce Nexys Video Artix-7 FPGA . . . 62
3.4 Ukázkový SoC pro DE1-SoC 63
3.5 DE1-SoC s připojeným UARTem 63
3.6 Rozložení SoC v FPGA na desce DE1-SoC 64

Seznam tabulek

1.1 Porovnání vývojových FPGA desek 18

2.1 Přístup k paměťovým oblastem procesoru Cortex-M0 [19] 22
2.2 Použité rozložení paměti 22
2.3 Datové nároky na komunikaci 23
2.4 Propustnost procesoru při použití jedno, dvou a čtyřbajtových transakcí 23

3.1 Výsledek implementace pro vývojovou desku Nexys Video Artix-7 FPGA 60
3.2 Výsledek implementace pro vývojovou desku DE1-SoC 61

Úvod

Systém na čipu (SoC) je integrovaný obvod, který obvykle obsahuje jeden či více procesorů, paměti, speciální bloky pro akceleraci určitých funkcí a periferie, které komunikují s okolním světem. Své uplatnění nachází typicky ve vestavných systémech.

První kapitola (Analýza) obsahuje rozbor používaných procesorů, sběrnic a sériových rozhraní, se kterými je možno se v SoC obvodech setkat.

Druhá kapitola pojednává o samotné realizaci a popisuje jednotlivé komponenty SoC, firmware a generátor.

Cílem práce bylo vyvinout nástroj v jazyce *Python*, který umožní vytváření systémů na čipu podle zadaných parametrů. Nástroj umožňuje další rozšiřování (snadné přidávání podpory pro další periferie, sběrnice apod.).

Vytvářené SoC je založeno na procesoru *ARM Cortex-M0* a integruje periferie třetích stran (např. *UART*, *SPI*, I^2C a *GPIO*). SoC je modelováno na úrovni RTL v jazyce *Verilog*.

Jedním z možných využití vyvinutého systému je vytváření SoC, sloužících k validaci ASIC obvodů. SoC se vytvoří s periferiemi potřebnými pro připojení daného ASICu, který má být validován. Pro ovládání (přístup do registrové mapy a/nebo přenos dat) ASICu se obvykle využijí rozhraní *UART*, *SPI* nebo I^2C. Ale rovněž půjdou přidat periferie specifické pro daný ASIC, buď vlastní nebo třetích stran. Samotná validace pak proběhne pomocí PC připojeného k SoC.

Vytvořený SoC je možné implementovat v FPGA. Na rozdíl od SoC vytvořených pomocí nástrojů dostupných v prostředích *Quartus* [1] a *Vivado* [2] není závislý na konkrétním výrobci a je možné ho implementovat v FPGA od největších výrobců (*Xilinx* a *Intel* (*Altera*)).

Dalším cílem je vytvořit univerzální firmware v jazyce C pro vytvořený SoC, který umožní ovládání všech periferií prostřednictvím počítače. Komunikace s počítačem je na straně FPGA realizována pomocí rozhraní *UART*. Pro připojení k počítači je použit *USB/UART* převodník integrovaný na vývojové desce. Ovládání z PC je možné realizovat programem v jazyce *Python*

s použitím knihovny, která byla napsána v rámci diplomové práce pro usnadnění komunikace.

Poslední kapitola se zabývá verifikací, která umožňuje ověřit správné propojení a funkci SoC a firmwaru pomocí simulace, a validací pomocí implementace v FPGA.

Kapitola 1

Analýza

Tato kapitola se zabývá existujícími generátory SoC a dále rozborem používaných procesorů, sběrnic a sériových rozhraní, se kterými je možno se v SoC obvodech setkat.

1.1 Generátory SoC

1.1.1 Vivado Design Suite IP Integrator (Xilinx)

Vivado Design Suite IP Integrator je nástroj od firmy *Xilinx* a je součástí prostředí *Vivado* [2]. Umožňuje vytvářet systémy pomocí propojování předem připravených IP jader. Vytvářet a propojovat jednotlivé bloky je možné pomocí grafického rozhraní (obr. 1.1), popř. programově použitím skriptovacího jazyka *Tcl.* [3]

Přímé propojování bloků s různými sběrnicemi není možné, je nutné explicitně vytvořit most a potřebné propojení provést ručně.

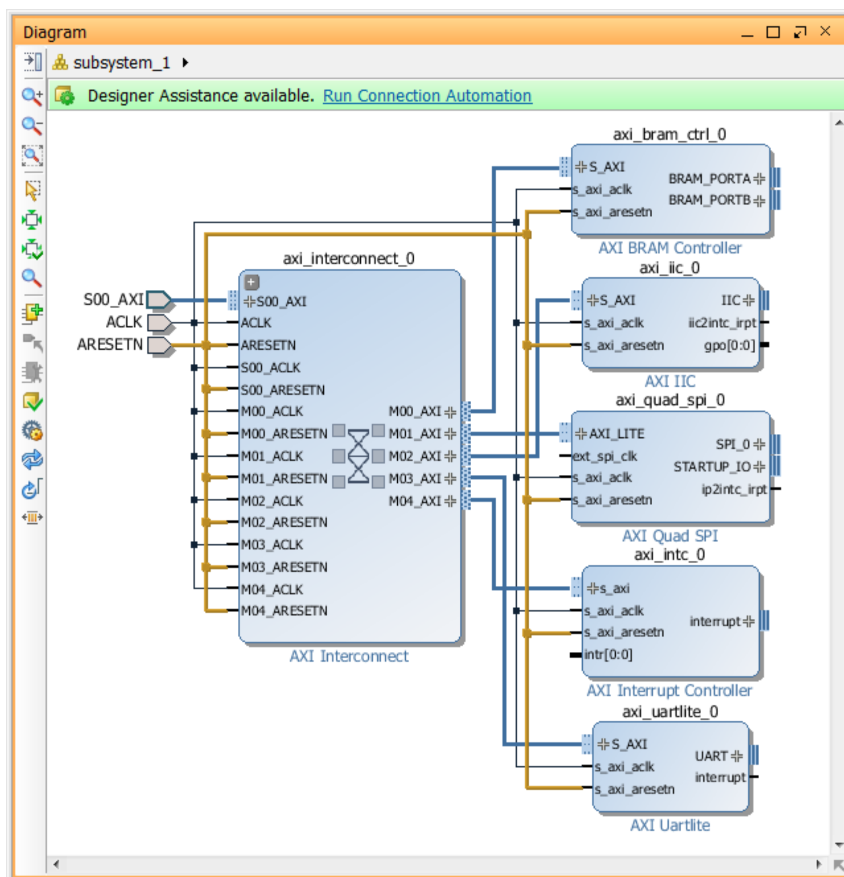
Do návrhového nástroje je možné přidávat vlastní IP jádra, nicméně podpora jiných sběrnic než *AXI4*, kterou používá většina integrovaných periferií, je omezená. Přidávání periferií s jinou sběrnicí než *AXI4* je proto složité.

Vytvořený návrh je možné použít pouze na FPGA *Xilinx*.

1.1.2 Qsys (Intel (Altera))

Qsys (obr. 1.2) je obdobný nástroj jako *Vivado Design Suite IP Integrator*, ale je určený pro FPGA *Intel* (dřívější *Altera*). Je součástí prostředí *Quartus* [1]. Rovněž lze použít předem připravená IP jádra. Všechna integrovaná jádra používají sběrnici *Avalon*, ale nástroj umožňuje pracovat i se sběrnicemi *AMBA* (*AXI*, *AHB* a *APB)*, které je možné použít při vytváření vlastních periferií.

Nástroj dokáže automaticky vytvořit propojení mezi různými sběrnicemi, není tedy třeba vytvářet potřebné mosty ručně.



Obrázek 1.1: Návrh v nástroji Vivado Design Suite IP Integrator

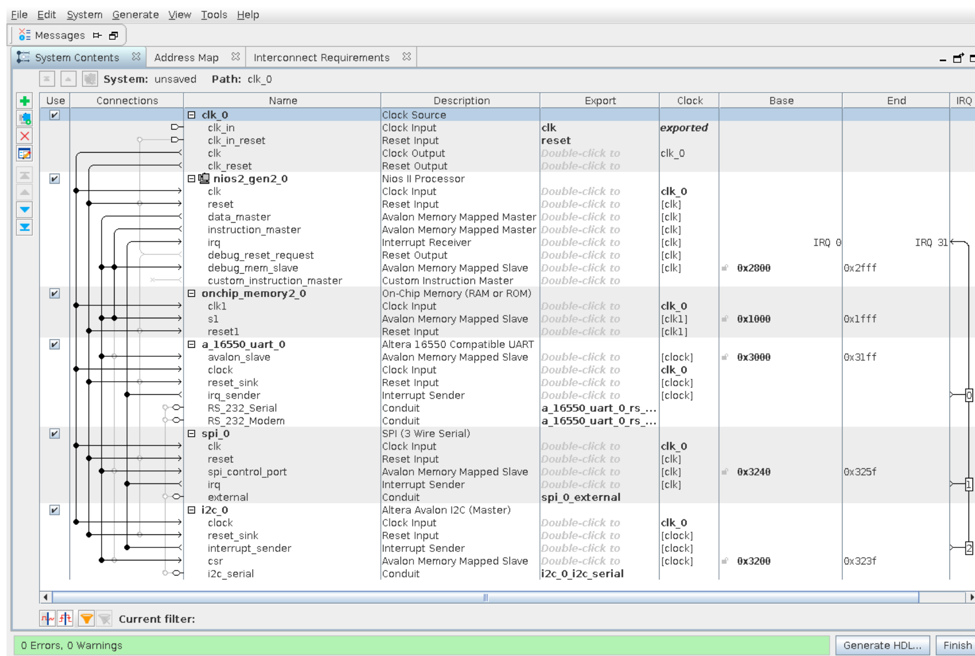
Stejně jako u předchozího nástroje je i v tomto případě možné vytvořený návrh použít pouze na FPGA *Intel*.

1.2 Procesory

Základem každého systému na čipu je procesor. Pro FPGA existuje množství procesorů od malých a jednoduchých až po složité, které umožňují běh operačních systémů.

1.2.1 MicroBlaze

MicroBlaze je 32b RISC procesor od firmy *Xilinx* optimalizovaný pro FPGA, ale lze jej licencovat i pro použití v *ASIC*. Procesor je konfigurovatelný a umožňuje přidávat funkce pro zrychlení určitých instrukcí (násobičku, děličku apod.). Pro připojení instrukčních a datových pamětí používá sběrnici *LMB*, pro připojení ostatních komponent sběrnici *AXI*. [4]



Obrázek 1.2: Ukázka SoC v nástroji Qsys

Vlastnosti:

- 32 32b registrů,
- 32b instrukce,
- 32b adresní sběrnice,
- oddělená adresní a datová sběrnice,
- volitelná podpora MMU umožňující běh operačního systému (např. *Linux*).

Microblaze je součástí vývojového prostředí *Vivado*, dostupný pouze pro FPGA firmy *Xilinx*. Programy pro *Microblaze* lze vytvářet pomocí prostředí *Xilinx SDK*.

1.2.2 Nios II

Nios II je procesor vytvořený firmou *Altera* pro jejich FPGA. Je srovnatelný s MicroBlaze. Existují dvě varianty procesoru: *Nios II/f (fast)* a *Nios II/e (economy)*. První je zaměřen na vysoký výkon a umožňuje přidávat jednotky pro správu paměti (MPU a MMU), hardwarové násobičky a děličky a instrukční i datovou cache. Varianta *economy* je zaměřena na co nejmenší plochu.

Vlastnosti [5]:

- plně 32b instrukční sada, datová i adresní sběrnice,
- 32 registrů,
- až 32 zdrojů přerušení (další pomocí externího řadiče přerušení),
- volitelná podpora MMU umožňující běh operačního systému (např. *Linux*),
- možnost přidávat vlastní instrukce.

1.2.3 OpenRISC

Projekt *OpenRISC* si klade za cíl vytvořit svobodný procesor pro použití ve vestavných systémech. První (a jediná) architektura byla zveřejněna v roce 2000 pod jménem *OpenRISC 1000*. Poslední revize vyšla v roce 2014. Architektura popisuje instrukční sadu procesoru (ISA), která používá 32b instrukce (všechny stejné délky), počítá s 32b nebo 64b adresním prostorem a definuje tyto skupiny instrukcí [6]:

- ORBIS32/64: Základní instrukce (load/store, aritmetické, logické, ...) pracující s 32-bit/64-bit daty,
- ORVDX64: Vektorové/DSP rozšíření (SIMD) pracující s 8, 16, 32 a 64b daty,
- ORFPX32/64: Rozšíření pro práci s čísly v plovoucí desetinné čárce (32b a 64b).

V současnosti existuje několik implementací této architektury (OR1200, mor1kx aj.). Tyto implementace používají sběrnici *Wishbone*.

1.2.4 ARM Cortex-M

Rodina procesorů *ARM Cortex-M* je navržena pro použití v mikrořadičích. Všechny procesory používají RISCovou instrukční sadu *Thumb* a *Thumb-2*. Podporované instrukce se však u jednotlivých modelů liší. Vyšší modely podporují ochranu paměti pomocí MPU.

Procesor *ARM Cortex-M0* je podrobněji popsán v kapitole 2.2.1.

1.3 Sběrnice

1.3.1 AMBA

AMBA (Advanced Microcontroller Bus Architecture) je otevřený standard, který popisuje sběrnice určené pro komunikaci uvnitř čipu. Standard byl zveřejněn v roce 1996, od té doby vyšlo několik revizí, které upravují jednotlivé sběrnice (přidávají další funkce) a přidávají další sběrnice.

V současnosti se používají tyto sběrnice [7]:

- *CHI (Coherent Hub Interface)* – má nejvyšší výkon, používá se pro sítě a servery,
- *ACE (AXI Coherency Extensions)* – používá se v systémech obsahujících *ARM big.LITTLE*^{TM} (chytré telefony, tablety apod.),
- *AXI (Advanced eXtensible Interface)* – nejrozšířenější sběrnice, dokáže propojit stovky bloků v SoC,
- *AHB (Advanced High-Performance Bus)* – hlavní systémová sběrnice pro použití v mikrokontrolérech,
- *APB (Advanced Peripheral Bus)* – jednoduchá sběrnice určená pro periferie,
- *ATB (Advanced Trace Bus)* – sběrnice určená pro ladění pomocí *ARM CoreSight*^{TM}.

1.3.1.1 AXI

Sběrnici *AXI* je možno použít v systémech vyžadujících vysoký výkon a frekvenci. Protokol *AXI* [8]:

- má vysokou propustnost a nízkou latenci,
- umožňuje použití vysokých frekvencí bez použití složitých mostů,
- splňuje požadavky širokého množství komponent,
- je vhodný pro paměťové řadiče s vysokou počáteční latencí,
- může být použit spolu s rozhraními *AHB* a *APB*.

Hlavní vlastnosti *AXI* [8]:

- oddělené adresní/řídící a datové fáze,
- podporuje nezarovnané přenosy,
- používá blokové přenosy,

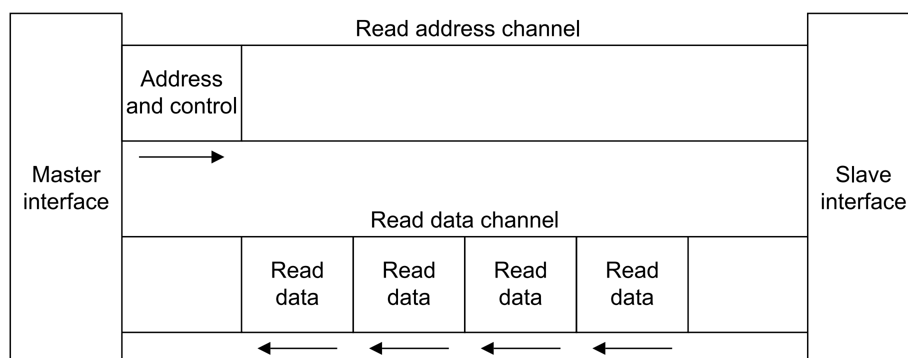
- oddělené kanály pro zápis a čtení,
- umožňuje přidání dalších registrů pro zvýšení pracovní frekvence.

Komunikace probíhá pomocí pěti (do jisté míry) nezávislých kanálů. Dva jsou určeny pro čtení (obr. 1.3), zbylé tři pro zápis (obr. 1.4).

Kanály:

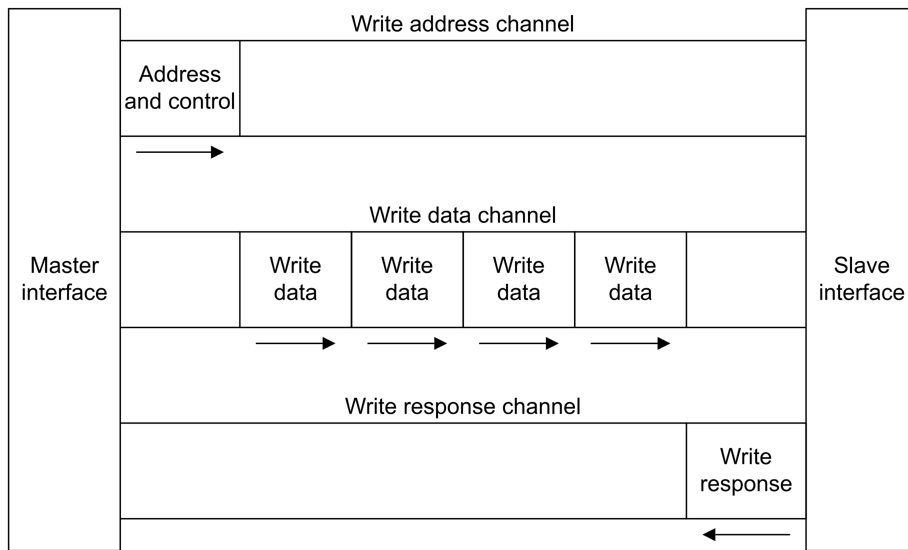
- čtení – adresa
- čtení – data
- zápis – adresa
- zápis – data
- zápis – odpověď

Pro čtení a zápis jsou použity oddělené adresní kanály. Spolu s adresou jsou přenášeny také všechny řídící informace. Datové kanály jsou rovněž oddělené, čtení a zápis tak může probíhat zároveň.

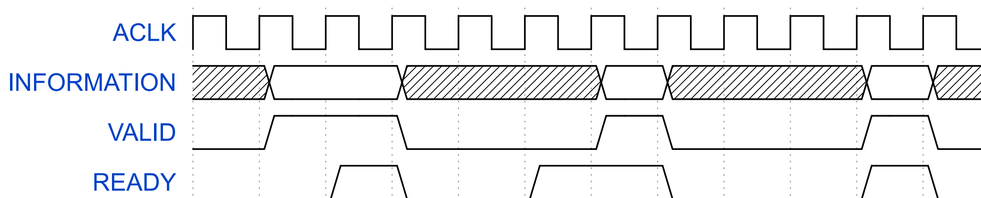


Obrázek 1.3: Kanály sběrnice AXI při čtení [8]

Každý kanál používá pro signalizaci platnosti a potvrzování dat signály „VALID“ a „READY“. Zdroj nastavuje signál „VALID“, čímž indikuje platnost dat na sběrnici. Cíl nastavuje signál „READY“, jakmile je připraven data přijmout. Přenos nastane pouze tehdy, když jsou oba signály nastaveny. Jak cíl, tak zdroj tedy můžou kanál pozdržet. Různé příklady zachycuje obrázek1.5.



Obrázek 1.4: Kanály sběrnice AXI při zápisu [8]



Obrázek 1.5: Různé příklady potvrzování dat v kanálu [8]

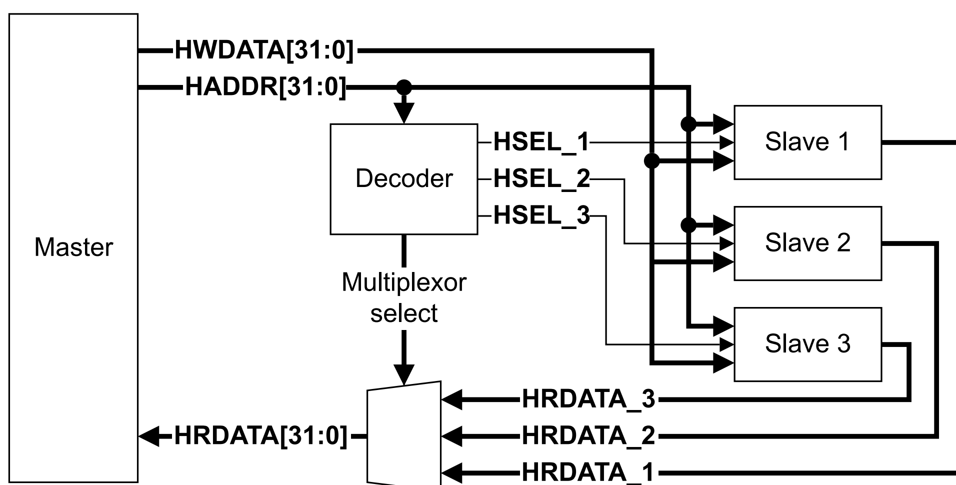
1.3.1.2 AHB

AHB se používá jako hlavní systémová sběrnice v mikrokontrolérech. K *AHB* jsou připojeny interní a externí paměti a periferie vyžadující vysokou propustnost. Ačkoliv mohou být všechny periferie připojeny ke sběrnici *AHB*, z výkonnostních důvodů se periferie nevyžadující vysokou propustnost připojují ke sběrnici *APB*, která je k hlavní části systému připojena pomocí mostu. [9]

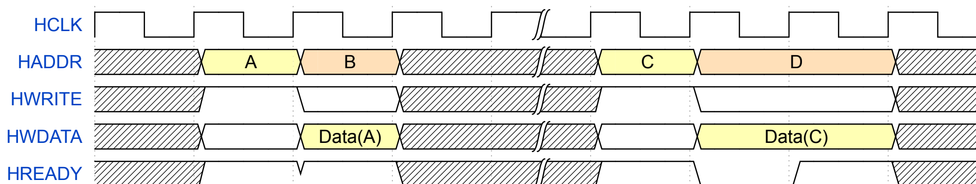
Obrázek 1.6 zobrazuje systém, kde je ke sběrnici připojena trojice zařízení. Propojení je realizováno pomocí adresního dekodéru, který podle adresy vybírá zařízení, se kterým probíhá komunikace, a multiplexoru, přes který se vrací data z vybraného zařízení.

Každá transakce se skládá ze dvou fází, adresní a datové. Během adresní fáze nastaví master adresu a řídící signály, které určují např. šířku a typ přenosu, v další fázi pak samotná data. Pokud zařízení nestíhá transakci zpracovat, může prodloužit datovou fázi transakce (obr. 1.7 a 1.8).

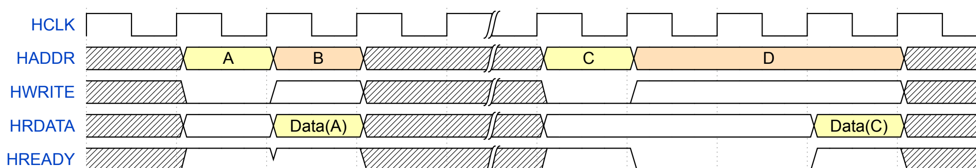
Transakce na sběrnici jsou prokládané, což znamená, že v jednom okamžiku probíhá adresní fáze jedné transakce a zároveň datová fáze druhé transakce.



Obrázek 1.6: Sběrnice AHB [9]



Obrázek 1.7: Zápis na sběrnici AHB (bez a s prodlouženou datovou fází) [11]



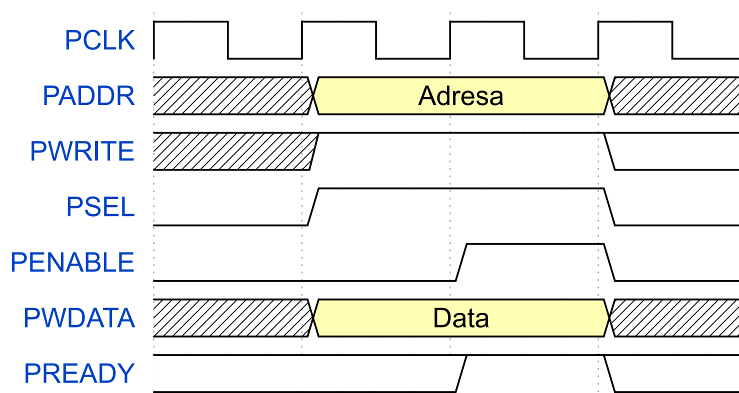
Obrázek 1.8: Čtení na sběrnici AHB (bez a s prodlouženou datovou fází) [11]

AHB-Lite je odlehčená verze sběrnice přidaná ve třetí revizi *AMBA*. Je určená pro systémy, které vyžadují vysoký výkon, ale obsahují pouze jediné zařízení typu master. Pokud je potřeba takovýto systém rozšířit o podporu více zařízení typu master, použije se zapojení *Multi-layer AHB* [10] (popsané více v kapitole 2.2.2.3)

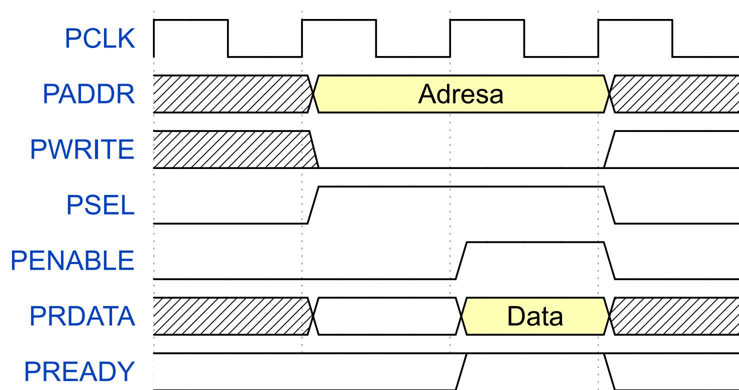
1.3.1.3 APB

APB se používá k připojení periferií, které od sběrnice nevyžadují vysokou propustnost.

Každá transakce zabere alespoň dva cykly, ale na rozdíl od *AHB* nejsou transakce prokládané. V prvním cyklu jsou nastaveny pomocné signály, adresa a data (v případě zápisu). V druhém je nastaven signál „PENABLE“ a dochází k samotnému zápisu nebo čtení, všechny ostatní signály se přitom nemění (obr. 1.9 a 1.10). [12]

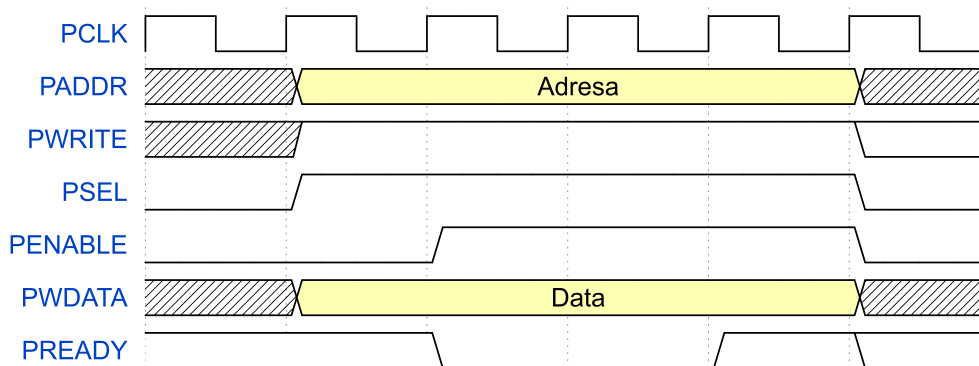


Obrázek 1.9: Zápis na sběrnici APB [12]

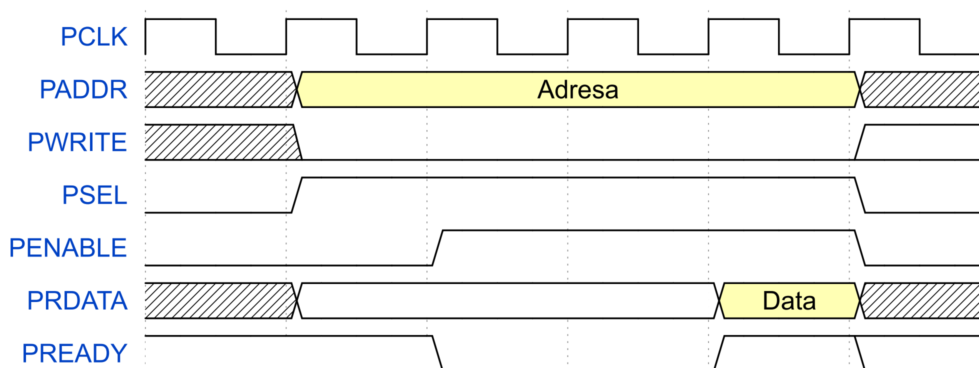


Obrázek 1.10: Čtení na sběrnici APB [12]

Ve třetí revizi *AMBA* byla do *APB* začleněna dvojice signálů „PREADY“ a „PSLVERR“, které umožňují periferii vkládat čekací cykly nebo indikovat chybu při přenosu (obr. 1.11 a 1.12).



Obrázek 1.11: Zápis na sběrnici APB se dvěma čekacími cykly [12]



Obrázek 1.12: Čtení na sběrnici APB se dvěma čekacími cykly [12]

1.3.2 Avalon

Sběrnice *Avalon* zjednodušuje návrh systémů a umožňuje jednoduché propojování komponent v FPGA firmy *Altera*. Sběrnice definuje rozhraní, která jsou vhodná pro vysokorychlostní přenos dat, čtení a zápis z/do registrů a pamětí a ovládání externích zařízení. [13]

1.3.3 Wishbone

Wishbone je otevřená sběrnice určená pro použití v SoC, za kterou stojí projekt *OpenCores*. Důvodem pro vznik této sběrnice byla potřeba zavedení stejného rozhraní pro IP jádra. To umožňuje jejich snazší a rychlejší integraci do nových projektů. [14]

Signály jsou podobné jako u sběrnice *Avalon* (most mezi těmito sběrnicemi je pouze kombinační).

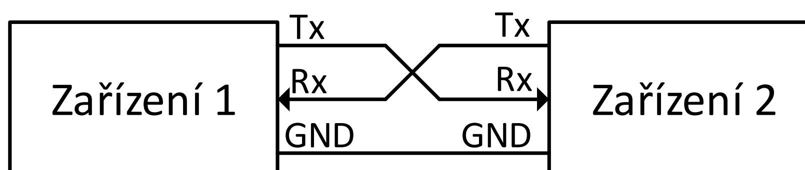
Wishbone tak používá mnoho otevřených jader, z nichž jsou některá dostupná přímo na stránkách projektu *OpenCores*.

1.4 Sériová rozhraní

1.4.1 UART

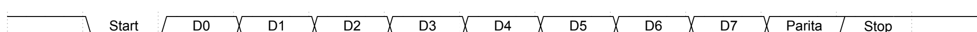
Jedná se o asynchronní plně duplexní rozhraní (existuje však i synchronní varianta - *USART*). Komunikace probíhá prostřednictvím dvou vodičů. *Rx* pro příjem a *Tx* pro odesílání.

Řízení datového toku je možné realizovat přidáním dalších vodičů (*RTS* a *CTS*), popř. softwarovou variantou. V praxi je však nejčastější dvouvodičová varianta bez jakéhokoliv řízení.



Obrázek 1.13: Ukázka zapojení UARTu

Zahájení komunikace probíhá vysláním *START* bitu (log. úroveň 0). Následně jsou vysílána data od nejméně významného bitu k nejvýznamnějšímu. Počet datových bitů není pevně daný, ale nejčastěji se jich používá osm. Data můžou být zabezpečena volitelným paritním bitem. Přenos se ukončuje pomocí *STOP* bitu (log. úroveň 1).



Obrázek 1.14: Ukázka komunikace UARTu

Obě zařízení musí mít před začátkem komunikace nastavenou stejnou rychlost a formát přenosu (počet datových bitů, parita). Rychlost obou zařízení by se neměla lišit o více než 2 %, jinak začne docházet k chybám při přenosu.

Rychlost přenosu může být libovolná, avšak zařízení podporují obvykle jen určitou množinu. Mezi standardní rychlosti patří: 9600, 19200, 38400, 57600, 115200 baudů.

1.4.2 SPI

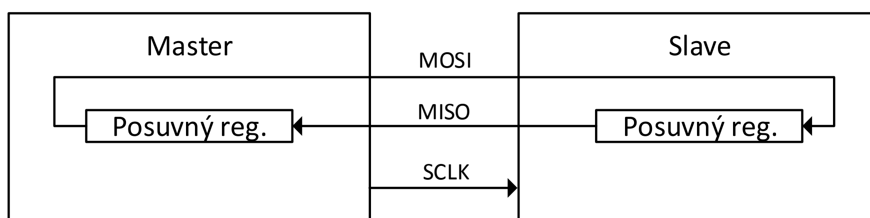
SPI (Serial Peripheral Interface) je synchronní sériové rozhraní typu master/slave. Slouží k připojení pamětí A/D a D/A převodníků a jiných zařízení.

Data se přenáší pomocí dvojice vodičů (pro každý směr jeden), platnost dat je určena vodičem přenášejícím hodinový signál, který je generován masterem. K rozhraní *SPI* může být připojeno více zařízení, která jsou vybrána pomocí zvláštních vodičů.

Vodiče rozhraní *SPI*:

- *MOSI (Master Out, Slave In)* – datový výstup (z pohledu zařízení v režimu master),
- *MISO (Master In, Slave Out)* – datový vstup (z pohledu zařízení v režimu master),
- *SCLK (Serial Clock)* – hodinový signál,
- *SS (Slave Select)* – slouží k výběru zařízení, někdy také označován jako *CS (Chip Select)*, většinou aktivní v log. nule.

Rozhraní *SPI* je založeno na posuvném registru, který v sobě má každé *SPI* zařízení (ať master nebo slave). Registry dvou zařízení jsou spolu propojeny tak, aby tvořily kruh (obr. 1.15). Oba registry se vždy posouvají současně (jsou řízeny pomocí signálu *SCLK*, který vytváří master) a s každým posunem si oba registry vymění jeden bit. Posuny pokračují, dokud se nevymění obsahy obou registrů.

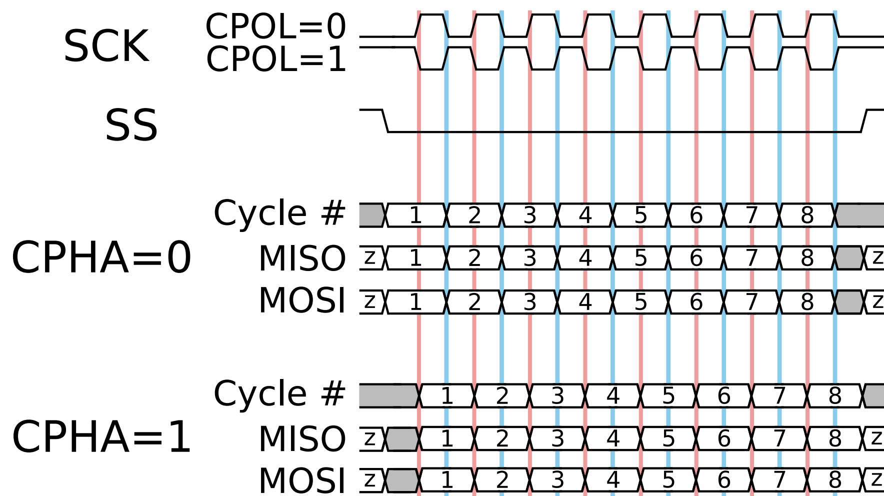


Obrázek 1.15: Posuvné registry rozhraní SPI

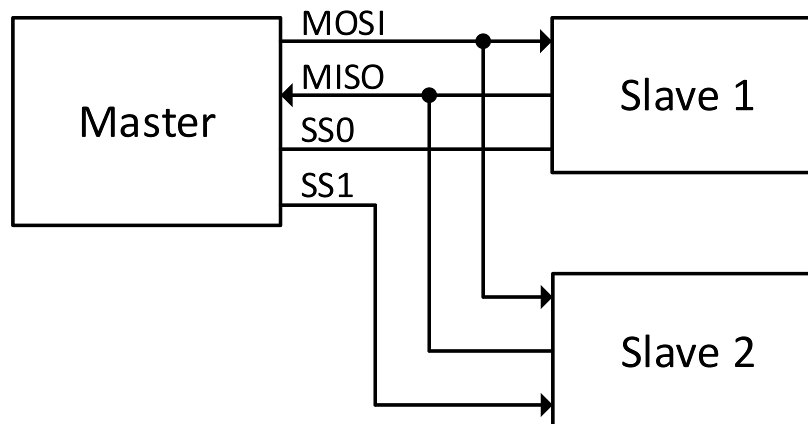
Rychlost komunikace je určena zařízením v roli master (obvykle však maximálně 25 MHz), musí však být menší než maximální podporovaná rychlost zařízení v roli slave. Master většinou umožňuje nastavit polaritu hodinového signálu pomocí konfiguračního bitu *CPOL* (u zařízení v roli slave může být režim nastaven napevno). Polarita určuje hodnotu hodinového signálu v klidové úrovni. Konfigurační bit *CPHA* označuje, při které hraně hodinového signálu

dochází k posunutí posuvných registrů (pokud je bit *CPOL* nastaven, dochází i k otočení smyslu hrany). [15]

Všechny možné kombinace konfiguračních bitů *CPOL* a *CPHA* zachycuje obrázek 1.16.



Obrázek 1.16: Čtyři možné průběhy signálů SPI [16]



Obrázek 1.17: Ukázka zapojení rozhraní SPI

1.4.3 I^2C

I^2C je dvouvodičové rozhraní typu master/slave vyvinuté firmou *Philips*, někdy z licenčních důvodů označované jako *TWI (Two Wire Interface)*. Je určeno pro připojení pomalejších periferií.

K *I^2C* je možné připojit více zařízení a to jak v roli slave, tak v roli master. Pokud na sběrnici chce komunikovat více zařízení typu master najednou, dochází k arbitraci.

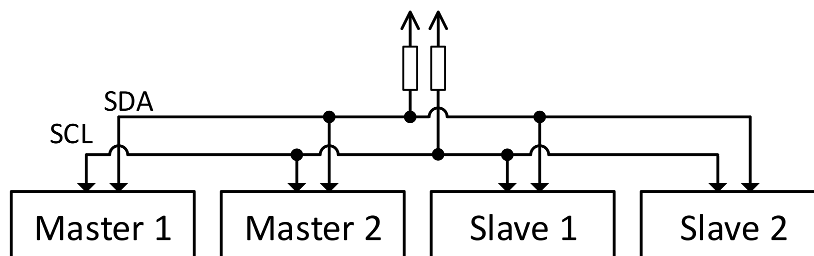
Ke komunikaci využívá dvojici vodičů, které používají zapojení s otevřeným kolektorem. K oběma vodičům musí být připojen externí pull-up rezistor. První vodič (*SCL*) je určen pro hodinový signál, který je vytvářen zařízením (vždy master), které chce komunikovat. Po druhém vodiči (*SDA*) jsou přenášena data.

Standard *I^2C* definuje několik rychlostí [17]:

- *Standard-mode (Sm)*, rychlost až 100 kbit/s,
- *Fast-mode (Fm)*, rychlost až 400 kbit/s,
- *Fast-mode Plus (Fm+)*, rychlost až 1 Mbit/s,
- *High-speed mode (Hs-mode)*, rychlost až 3.4 Mbit/s,
- *Ultra Fast-mode (UFm)*, rychlost až 5 Mbit/s (není zpětně kompatibilní s ostatními standardy).

Rychlost závisí na zařízení v roli master, které generuje hodinový signál, zařízení v roli slave však může komunikaci pozdržet přidržením hodinového signálu v log. nule.

Každé zařízení na sběrnici má přidělenou svoji adresu (buď 7 nebo 10 b). Přenos probíhá tak, že se po sběrnici nejprve pošle adresa zařízení spolu s informací o směru posílaných dat a poté následují samotná data.



Obrázek 1.18: Ukázka zapojení rozhraní I^2C

1.5 Vývojové FPGA desky

Tabulka 1.1 porovnává vývojové desky, které jsou v současnosti na trhu. Zaměřuje se na levnější desky s větším počtem IO. Všechny desky mají integrovaný převodník *USB/UART* (kromě *Genesys Virtex-5 FPGA*, která má místo převodníku konektor *RS-232*).

Vývojová deska	FPGA	IO na desce	Druh IO konektoru	IO v konekt.	IO celkem	Orientační cena
Xilinx:						
Cmod A7	Artix-7 (XC7A35T-1CPG236C)	44	-	-	44	$89
Arty Board Artix-7	Artix-7 (XC7A35T-L1CSG324I)	40	Arduino	43	83	$99
XuLA2-LX25	Spartan-6 (XC6LX16-CS324)	33	-	-	33	$119
Nexys 3 Spartan-6	Spartan-6 (XC6LX16-CS324)	32	VHDC	40	72	$270
Nexys Video Artix-7 FPGA	Artix-7 (XC7A200T-1SBG484C)	24	FMC	68	92	$490
ZedBoard Zynq-7000	Zynq-7000 (XC7Z020-CLG484)	40	FMC	68	108	$495
Genesys Virtex-5 FPGA	Virtex-5 (LX50T)	32	2x VHDC	80	112	$899
Genesys 2 Kintex-7	Kintex-7 (XC7K325T-2FFG900C)	40	FMC	160	200	$1299
Altera:						
SocKit	Cyclone V (5CSXFC6D6F31C6N)	0	HSMC	80	80	$350
Cyclone V GX Starter Kit	Cyclone V (5CGXFC5C6F27C7N)	36	HSMC	80	116	$179
DE1-SoC Board	Cyclone V (5CSEMA5F31C6)	72	-	-	72	$175
DE0-Nano-SoC Kit/ Atlas-SoC Kit	Cyclone V (5CSEMA4U23C6N)	72	Arduino	14	86	$99

Tabulka 1.1: Porovnání vývojových FPGA desek

Kapitola **2**

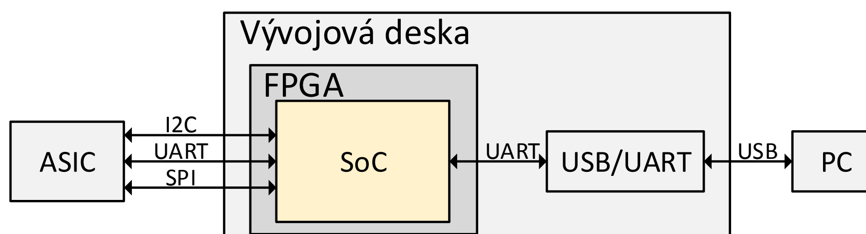
Realizace

Navržený SoC je implementován v FPGA a připojen k počítači pomocí převodníku *USB/UART* integrovaném na vývojové desce. SoC obsahuje periferie, pomocí kterých komunikuje s dalšími obvody. Může sloužit k validaci dalších ASIC obvodů (obr. 2.1).

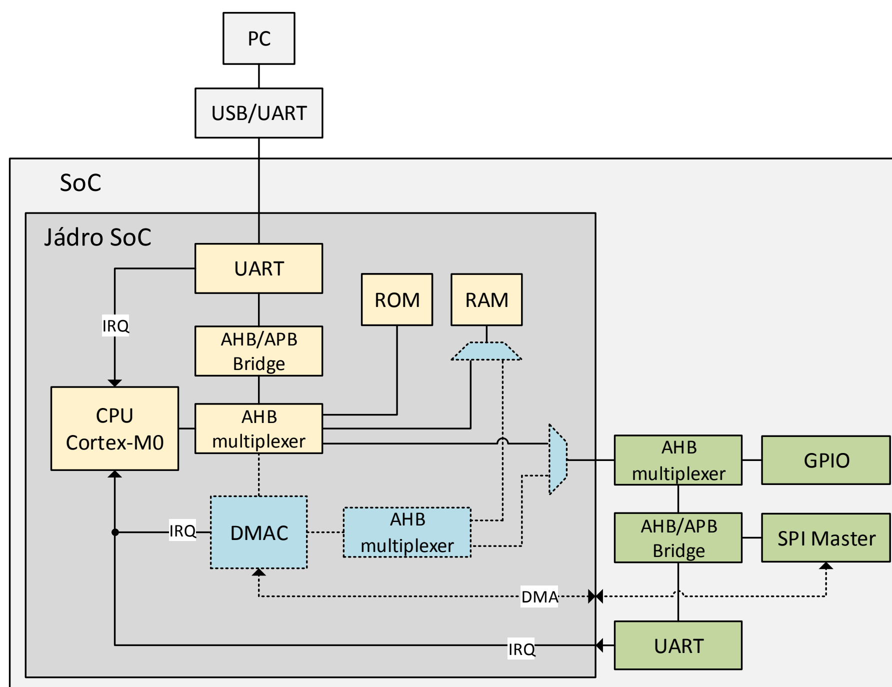
2.1 Architektura SoC

Navrhovaný systém na čipu se skládá z neměnného jádra, které obsahuje procesor *Cortex-M0*, paměti ROM (pro program) a RAM (pro data), *UART* (pro komunikaci s PC) a řadič DMA (obr. 2.2). Pomocí parametrů lze volit velikosti pamětí RAM a ROM. Procesor, *UART* a řadič DMA jsou *IP* třetích stran.

Z jádra je vyvedena sběrnice *AHB-Lite*, ke které jsou připojovány vygenerované periferie dle přání uživatele. Do jádra rovněž vedou signály přerušení a rozhraní pro DMA od vygenerovaných periferií.



Obrázek 2.1: Použití vytvořeného SoC pro validaci ASICu



Obrázek 2.2: Příklad vytvořeného SoC

Pokud DMA není potřeba (žádná z připojených periferií ho nevyžaduje), je řadič DMA z jádra pomocí parametru úplně odstraněn.

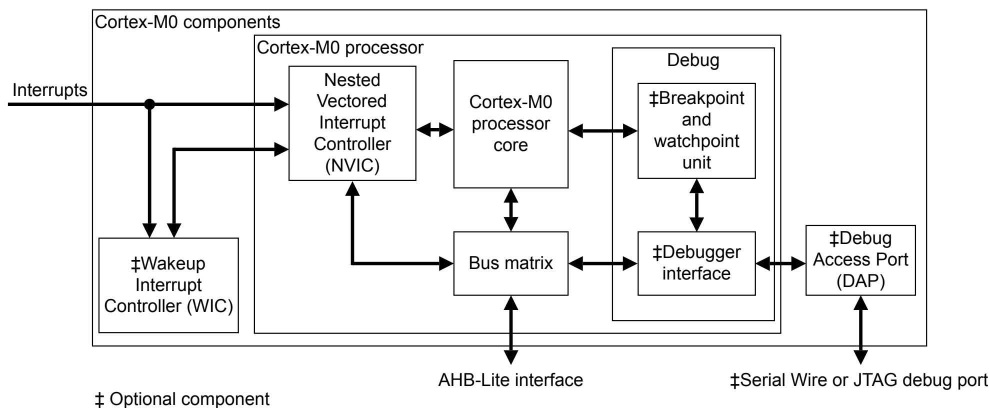
2.2 Komponenty

2.2.1 Procesor ARM Cortex-M0

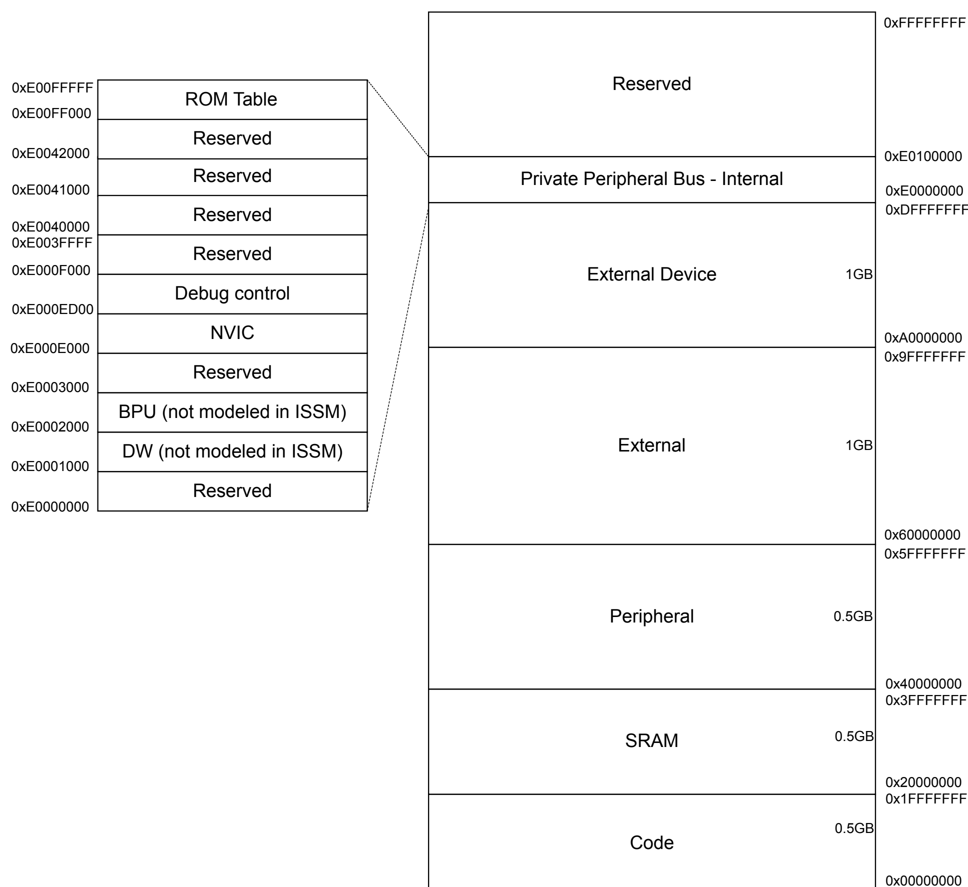
Pro realizaci byl vybrán *Cortex-M0* od firmy *ARM*. Jedná se o nejmenší procesor z rodiny *Cortex-M*. Používá sběrnici *AHB-Lite* (odlehčená forma sběrnice *AHB*). Součástí procesoru je také řadič přerušení. [18]

2.2.1.1 Paměťový prostor

Procesor *Cortex-M0* používá 32b (stejná šířka pro adresy i data) sběrnici. Používá Von Neumannovu architekturu, tedy sběrnice pro data a program je společná. Procesor tak může přistupovat ke 4 GB paměťového prostoru, do kterého je mapována paměť programu, operační paměť a jednotlivé periferie. Část paměťového prostoru je také vyhrazena pro řadič přerušení.



Obrázek 2.3: Funkční bloky procesoru Cortex-M0 [18]



Obrázek 2.4: Doporučené rozložení paměti procesoru Cortex-M0 [19]

Je vhodné zachovat doporučené rozdělení paměťového prostoru (obr. 2.4), protože procesor může pracovat s každou oblastí trochu jinak. Např. může načítat instrukce jen z určitých oblastí.

Rozlišují se tři druhy přístupu: „program", „data" a „zařízení" (tab. 2.1). Oblasti pro „program" a „data" jsou podobné (v případě procesoru *ARM Cortex-M0* se plně překrývají). Procesor při přístupu do těchto oblastí nastavuje příznak *bufferable*. To znamená, že prováděné transakce mohou být potvrzovány, i když nedošlo k samotnému zápisu dat. Naproti tomu je u oblasti pro „zařízení" takové chování nežádoucí, protože by při přístupu k registrům periferií způsobovalo problémy.

Adresní rozsah	Program	Data	Zařízení
0xF000_0000 – 0xFFFF_FFFF	Ne	Ne	Ano
0xE000_0000 – 0xEFFF_FFFF	Ne	Ne	Ne
0xA000_0000 – 0xDFFF_FFFF	Ne	Ne	Ano
0x6000_0000 – 0x9FFF_FFFF	Ano	Ano	Ne
0x4000_0000 – 0x5FFF_FFFF	Ne	Ne	Ano
0x2000_0000 – 0x3FFF_FFFF	Ano	Ano	Ne
0x0000_0000 – 0x1FFF_FFFF	Ano	Ano	Ne

Tabulka 2.1: Přístup k paměťovým oblastem procesoru Cortex-M0 [19]

U generovaného SoC by se však takovéto chování neprojevilo, protože v systému pro generování SoC není jediná komponenta, která by příznak *bufferable* používala. Nicméně při rozšíření systému o složitější mosty mezi sběrnicemi by určité problémy mohly nastat.

Použité rozložení paměti (tab. 2.2) tedy respektuje doporučení.

Adresní rozsah	Použití
0xA000_0000 – 0xBFFF_FFFF	Vygenerované periferie
0x4000_0000 – 0x4000_07FF	Periferie v „jádru"
0x2000_0000 – 0x200F_FFFF	Paměť RAM (data)
0x0000_0000 – 0x000F_FFFF	Paměť ROM (program)

Tabulka 2.2: Použité rozložení paměti

2.2.2 Sběrnice

2.2.2.1 Výpočet propustnosti

Při použití vysokorychlostních periferií (např. *SPI*) může mít procesor problémy přenést všechna potřebná data do a z periferie a v důsledku toho může docházet k výpadkům komunikace nebo dokonce ke ztrátě dat. Je tedy dobré si udělat alespoň hrubou představu o tom, kolik dat je schopen procesor přenést.

Nároky na datové přenosy zobrazuje tabulka 2.3.

periferie	SPI	UART	I^2C (High-speed)
frekvence [MHz]	20	0,9216	3,6
poloduplexní [MB/s]	2,5	0,09216	0,36
plně duplexní [MB/s]	5	0,018432	

Tabulka 2.3: Datové nároky na komunikaci

Měřením v simulaci bylo zjištěno, že procesor *Cortex-M0* je schopen přenést data z paměti do periferie (nebo naopak) každých devět hodinových taktů. Nezáleží přitom, jestli transakce přenáší jeden, dva nebo čtyři bajty. Naměřenou propustnost zachycuje tabulka 2.4. Jedná se o horní mez toho, co je procesor schopen přenést, skutečná přenosová rychlost je menší.

Z tabulky je vidět, že procesor pravděpodobně nebude mít problém s přenášením dat do periferií jako *UART* nebo I^2C. Problém může nastat při použití vysokorychlostních periferií jako je *SPI*, kdy procesor už nemusí stíhat potřebnou komunikaci obsloužit. V takových případech je lepší komunikaci realizovat prostřednictvím řadiče DMA, aby nezatěžovala procesor.

Frekvence procesoru [MHz]	Propustnost [MB/s]		
	1 B	2 B	4 B
30	3,33	6,67	13,33
40	4,44	8,89	17,78
50	5,56	11,11	22,22
60	6,67	13,33	26,67

Tabulka 2.4: Propustnost procesoru při použití jedno, dvou a čtyřbajtových transakcí

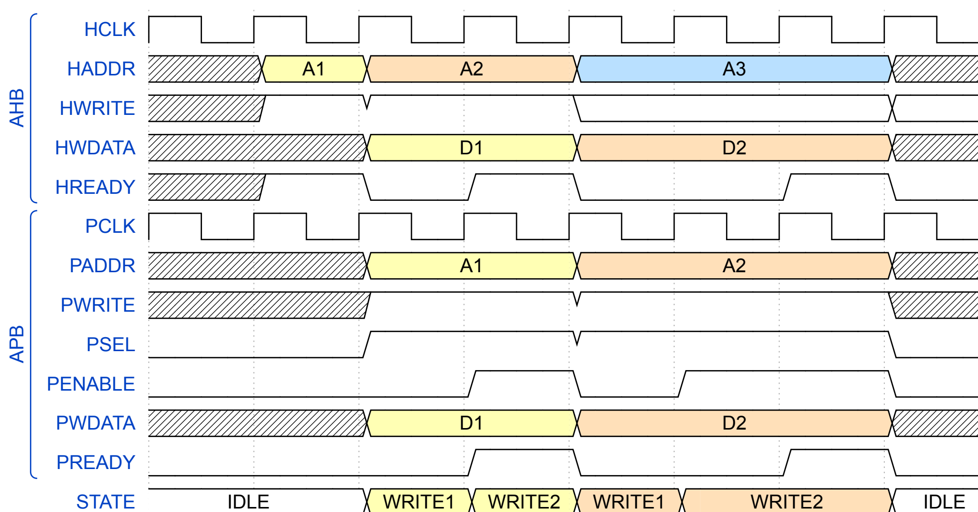
2.2.2.2 AHB/APB most

Většina použitých periferií má sběrnici *APB*, která je jednodušší než *AHB*. Pro převod sběrnice *AHB* na sběrnici *APB* byl v jazyce *Verilog* napsán most (angl. *bridge*).

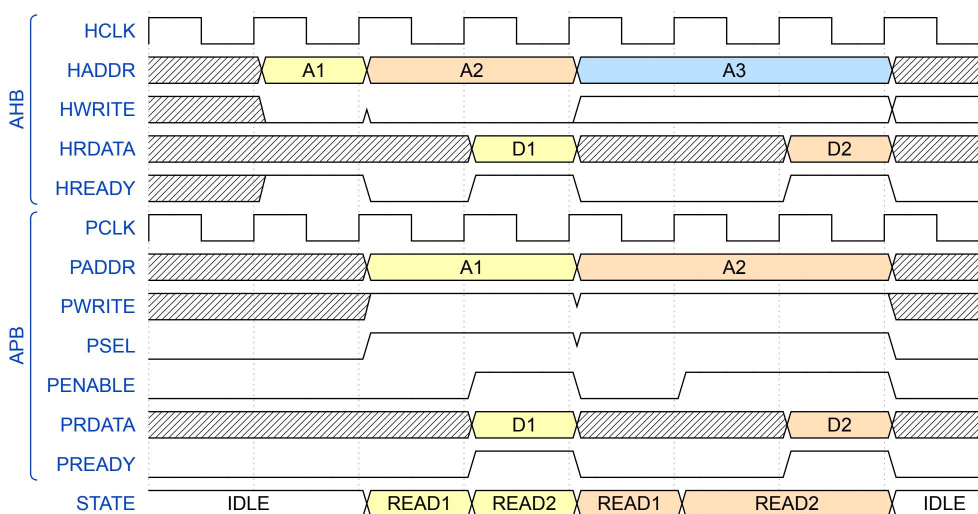
Vytvořený most převádí transakce z *AHB-Lite* na sběrnici *APB* (revize z *AMBA 3*). Každá transakce na sběrnici *APB* trvá alespoň dva cykly. Sběr-

nice *AHB-Lite* je proto při každé transakci pozdržena o jeden cyklus. Periferie však může transakci ještě prodloužit o libovolný počet cyklů.

Časové diagramy zachycující čtení a zápis přes most znázorňují obrázky 2.5 a 2.6.



Obrázek 2.5: Časový diagram zápisu do AHB/APB mostu

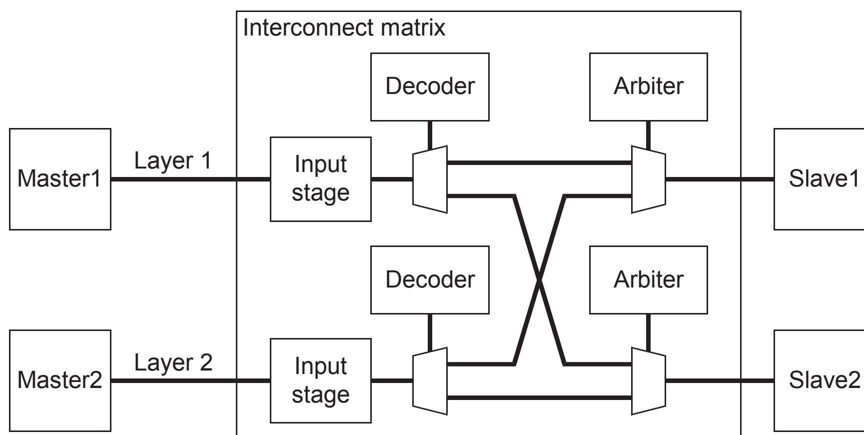


Obrázek 2.6: Časový diagram čtení z AHB/APB mostu

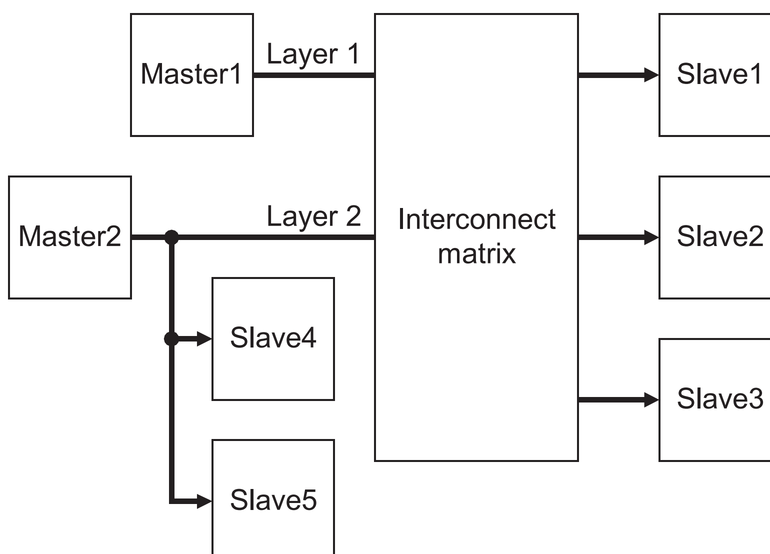
2.2.2.3 AHB arbitr

AHB-Lite na rozdíl od *AHB* neumožňuje použití více zařízení v roli master na jedné sběrnici. Tento problém řeší zapojení nazývané *Multi-Layer AHB*.

Každý master má svoji vlastní sběrnici (vrstvu), které jsou pomocí arbitru spojeny do jedné. Ta je následně připojena k zařízení (obr. 2.7). Každé zařízení může mít vlastní arbitr nebo k jednomu arbitru může být připojeno více zařízení. Některá zařízení nemusí být k arbitru připojena vůbec a můžou být dostupná pouze z jedné vrstvy (obr. 2.8).



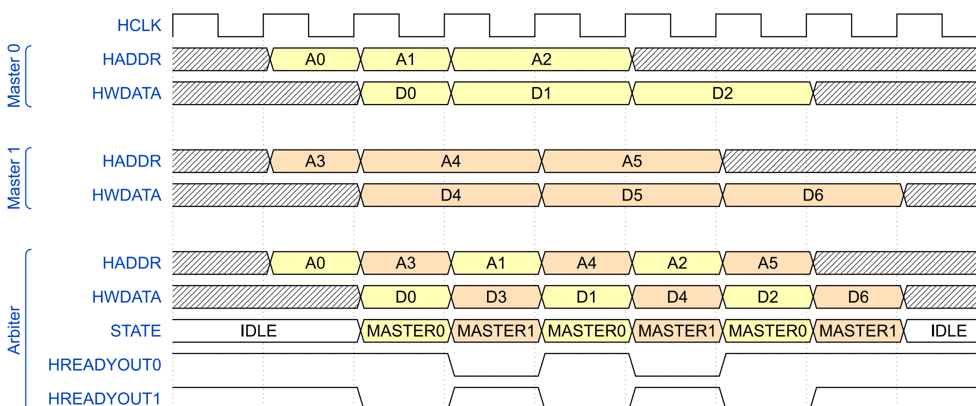
Obrázek 2.7: Zapojení AHB s arbitry [10]



Obrázek 2.8: Ukázka systému s více mastery [10]

Součástí balíčku s procesorem žádný arbitr nebyl, proto bylo potřeba vytvořit vlastní.

Vytvořený arbitr je napsán v jazyce *Verilog* a umožňuje připojení dvou vrstev *AHB-Lite*. Pokud arbitru přijdou požadavky z obou vrstev najednou, je jedna vrstva pozdržena a následně se vrstvy střídají pomocí metody *Round Robin* (obr. 2.9). Arbitr nepodporuje na výstupu blokové (angl. *burst*) transakce, dokáže je však přijmout a korektně překódovat na obyčejné.



Obrázek 2.9: Ukázka arbitrace

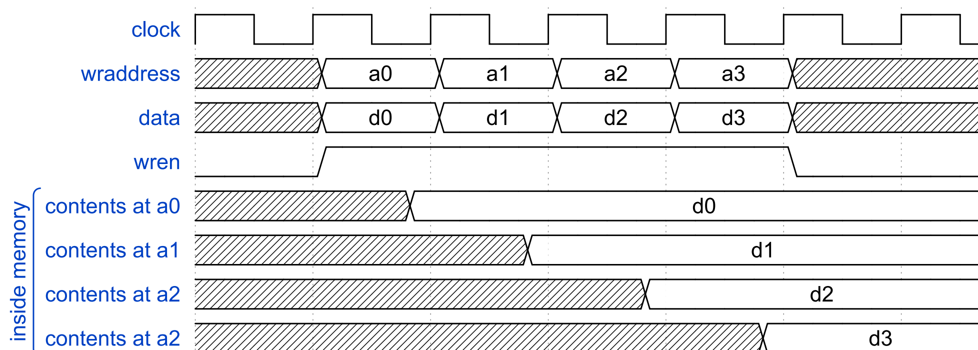
2.2.3 Paměti ROM a RAM

Paměti můžou v návrhu zabírat poměrně velkou část logiky. V FPGA je tento problém vyřešen pomocí vyhrazených paměťových bloků o velikosti jednotek až stovek KB. Těchto bloků, které jsou rozprostřené po celém čipu, může FPGA obsahovat až tisíce. Celková paměť obsažená v FPGA tak může sahat až k jednotkám MB. Z tohoto důvodu je v návrhu vhodné všechny větší paměti koncipovat tak, aby mohly tyto bloky využít. Toho je možné docílit použitím nástrojů dodávaných výrobci FPGA, výsledek však není přenositelný. Jako lepší řešení se tedy jeví modelovat paměť přímo na úrovni RTL tak, aby syntézní nástroje paměť správně poznaly a umístily ji do paměťových bloků.

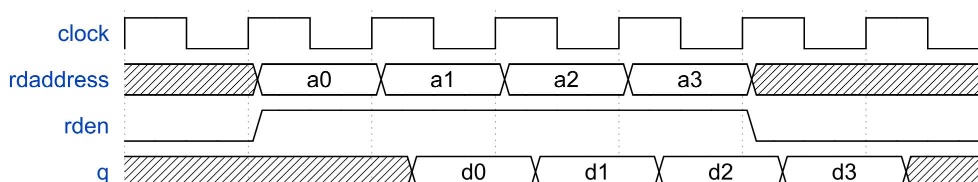
Paměťové bloky se z pohledu časování chovají obdobně jako klasické paměti typu SRAM s tím rozdílem, že mají oddělené vstupní a výstupní datové signály (obr. 2.10 a 2.11).

2.2.3.1 ROM

V paměti ROM je uložen program. Ze systému se do ní nedá zapisovat a její obsah tak zůstává neměnný. Paměť se v FPGA inicializuje při nahrání bitstreamu.



Obrázek 2.10: Zápis do blokové paměti FPGA [20]



Obrázek 2.11: Čtení z blokové paměti FPGA [20]

Připojení ke sběrnici *AHB-Lite* je triviální. Signály sběrnice je možné přímo připojit k signálům blokových pamětí. Jelikož je paměť určena jen pro čtení, je možné některé signály sběrnice *AHB-Lite* úplně ignorovat.

Inicializace paměti se v jazyce *Verilog* provede pomocí systémové procedury „$readmemh“ v bloku *initial*. Tato konstrukce je na FPGA syntetizovatelná.

```
1  assign hreadyout = 1'b1;
2  assign hresp     = 1'b0;
3
4  reg  [31:0] rom [0:4095];
5  wire [13:0] rom_index = haddr[15:2];
6
7  always @(posedge hclk)
8    if (hsel & hready & htrans[1])
9      hrdata <= {rom[rom_index][7:0], rom[rom_index][15:8],\
10      rom[rom_index][23:16], rom[rom_index][31:24]};
11
12 initial
13 begin
14   $readmemh("data.hex", rom);
15 end
```

Zdrojový kód 2.1: Ukázka 16KB paměti připojené ke sběrnici AHB-Lite

```
F0B59FB0
02900593
8B890F00
16001B06
```

Zdrojový kód 2.2: Inicializační soubor pro paměť o velikosti 4 slov po 32 b

2.2.3.2 Formát souboru pro $readmemh

Procedura „$readmemh“ očekává data uložena v textovém souboru, kde každý řádek reprezentuje jedno slovo paměti a jednotlivá slova jsou reprezentována číslicemi v šestnáctkové soustavě (znaky 0-9 a A-F).

2.2.3.3 RAM

Paměť RAM není potřeba na rozdíl od paměti ROM inicializovat, ale připojení ke sběrnici je složitější. Sběrnice *AHB-Lite* používá dělené transakce, nejdříve se na sběrnici objeví adresa a až poté data.

Čtení přes sběrnici *AHB-Lite* nepředstavuje problém, časově odpovídá chování pamětí v FPGA. Zápis je však komplikovanější. Paměti v FPGA předpokládají adresu i data pro zápis najednou. Je tedy nutné adresu na sběrnici zpozdit o jeden hodinový takt. Pokud by ale bylo potřeba po zápisu z paměti číst, nastala by kolize (adresa pro zápis a pro čtení by se měla posílat najednou).

Existuje několik řešení tohoto problému:

1. Čtení je možné v takovémto případě o jeden takt prodloužit. To by ale snížilo propustnost paměti.

2. Zápis přerušit, adresu i data uložit do bufferu a provést čtení z paměti. Následně data z bufferu zapsat do paměti při dalším zápisu. Pokud by zápis a následné čtení probíhalo ze stejné adresy, bylo by nutné data nečíst z paměti, ale z bufferu. Situace je ještě komplikovanější tím, že sběrnice *AHB-Lite* umožňuje přístup po slovech (32 b), půlslovech (16 b) a bajtech (8 b). Při zápisu např. půlslova a čtení celého slova ze stejné adresy by tak bylo nutné přečíst data z paměti a následně je zkombinovat s daty z bufferu.

3. Využít skutečnosti, že v FPGA jsou zpravidla dvoubránové paměti, což umožňuje provádět čtení a zápis současně.

Protože se práce zabývá vytvořením SoC pro FPGA, bylo použito poslední řešení.

2.2.4 Řadič DMA

Řadič DMA je volitelná součást vytvořeného SoC. Jeho použití dokáže ulehčit procesoru, pokud jsou potřeba větší datové přenosy.

Použitý řadič DMA pochází z knihovny *IP DesignWare* [21] od firmy *Synopsys*. Procesor komunikuje s řadičem pomocí sběrnice *AHB-Lite* (slave). Řadič má rovněž druhé rozhraní *AHB-Lite* (master), přes které probíhají datové přenosy.

Řadič DMA má přístup pouze do paměti RAM a k uživatelským periferiím. Pro spojení sběrnic vedoucích od procesoru a od řadiče jsou u paměti RAM a periferií vytvořeny arbitry.

2.2.4.1 Připojení periferií

Periferie komunikují s řadičem pomocí trojice signálů.

Signálem „dma_req“ periferie žádá o přenos bloku dat, velikost bloku určuje ovladač periferie. Signálem „dma_single“ periferie žádá o přenos dat pomocí jedné transakce. DMA řadič potvrzuje žádosti pomocí signálu „dma_ack“. Detailní popis rozhraní, včetně časování, je dostupný v [22].

2.2.5 UART, SPI Master, SPI Slave a I^2C

Použité periferie pochází z knihovny *IP DesignWare* od firmy *Synopsys*. Všechny periferie se připojují pomocí sběrnice *APB2*. Jedná se o plnohodnotné periferie, které obsahují hardwarové FIFO, dovolují nastavit rychlost a parametry přenosu a umožňují používání přerušení a řízení toku dat pomocí DMA. Detaily všech periferií jsou dostupné v [23] (UART), [24] (SPI) a [25] (I^2C).

2.2.6 GPIO

Periferie implementuje 16b *GPIO* a je součástí balíčku s procesorem *Cortex-M0*. Připojuje se pomocí sběrnice *AHB-Lite*.

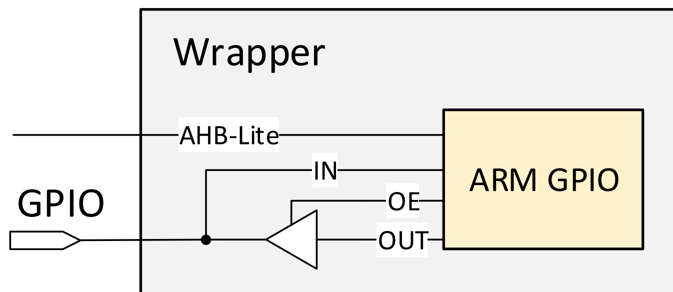
GPIO podporuje přerušení a připojování jiných periferií pomocí multiplexoru.

Nastavení pinů jako výstupních se provádí zápisem do registru „OUTENSET“. Pomocí zápisu do registru „OUTENCLR“ se piny nastaví jako vstupní. Samotná výstupní hodnota se nastaví zápisem do registru „DATAOUT“. Ke čtení hodnoty pinů se používá registr „DATA“.

Pro *GPIO* byl vytvořen ovladač, který umožňuje měnit směr, nastavovat a číst hodnotu jednotlivých pinů.

2.3 Firmware

Firmware zajišťuje komunikaci s počítačem a připojenými periferiemi. Skládá se z pevné části, která obsluhuje komunikaci s počítačem, a ovladačů periferií, které jsou přidány podle potřeby při sestavování firmwaru.



Obrázek 2.12: Obálka (wrapper) GPIO

Všechny ovladače implementují jednotné rozhraní, aby je bylo možné do firmwaru přidávat.

Se SoC je možné z PC komunikovat pomocí sady textových příkazů, které firmware interpretuje a v případě, že je příkaz určen některé z periferií, předá ho patřičnému ovladači, který má na starost samotnou komunikaci s periferií.

Příkazy a data jsou z počítače přijímána pomocí *UARTu*. Pokud by byla data přenášena přímo do periferie, znamenalo by to, že by rychlost periferií byla omezena rychlostí *UARTu*. Z toho důvodu používá firmware sadu bufferů (vyrovnávacích pamětí), jejichž počet a velikost si může uživatel nastavit.

Data z počítače se tedy nejprve ukládají do bufferu a až poté jsou dalším příkazem poslána do periferie. Obdobným způsobem probíhá také čtení. Data z periferie jsou nejprve ukládána do bufferu a až poté odeslána do počítače.

2.3.1 Ovladače periferií

Ovladač má na starosti přímou komunikaci s danou periferií. Se zbytkem firmwaru je propojen přes definované rozhraní.

Při startu firmwaru je zavolána inicializační funkce každého ovladače. Pomocí parametru je ovladači předána struktura s konfigurací (zdr. kód 2.3). Součástí struktury jsou informace pro ovladač. Např. adresa, na které se periferie nachází a informace o přerušeních a DMA. Druhá část struktury jsou ukazatele na funkce, které může ovladač vyplnit. Tyto funkce jsou volány při přijetí příkazů z počítače.

2.3.1.1 Popis rozhraní

Všechny ovladače musí použít hlavičkové soubory *peripherals.h* a *peripherals_init.h*, které definují rozhraní a předávají ovladači parametry pomocí maker.

```
typedef struct {
  void * const base_addr;    //adresa periferie
  const char * const name;   //jméno
  const uint32_t config_dma;  //informace o DMA
  const uint32_t config_intr; //informace o přerušeních
  const uint8_t index;        //číslo instance jedné periferie
  const uint8_t id;           //id (globálně unikátní)

  Peripheral_status (*set)(uint8_t index, const char *parameter,
    const char *value);
  Peripheral_status (*get)(uint8_t index, const char *parameter,
    char *data, uint32_t size);

  void (*read)(uint8_t index, uint8_t *addr, uint32_t size);
  void (*write)(uint8_t index, uint8_t *addr, uint32_t size);
  void (*stop)(uint8_t index, uint8_t *addr);

  void (*loop)(void);
  void (*intr)(uint8_t index, uint8_t intr);
  ...
} Peripheral;
```

Zdrojový kód 2.3: Konfigurační struktura ovladače periferie

Makro *SYS_FREQ* udává hodinovou frekvenci, na které pracují všechny periferie, ovladač může tuto informaci využít např. pro výpočet přenosových rychlostí.

Každý ovladač musí implementovat inicializační funkci. Jejím parametrem je ukazatel na konfigurační strukturu (zdr. kód 2.3). Nejdůležitější položka konfigurační struktury je „base_addr“. Ta označuje, na jakou adresu je mapována daná periferie.

Dalšími položkami jsou „config_intr“ a „config_dma“, které nesou informaci o povolených přerušeních a DMA. Tyto položky ovladač interpretuje jako bitový vektor, kde bit s hodnotou jedna označuje povolené přerušení nebo DMA. K jedné periferii tak může být připojeno až 32 signálů pro přerušení nebo 32 kanálů pro DMA.

V SoC může být více instancí stejné periferie, ovladač je ale pro stejné periferie vždy společný. Pro rozlišení jednotlivých instancí slouží položka „index“, která udává pořadové číslo periferie (číslováno od nuly). Celkový počet instancí (užitečný např. pro definování velikostí pomocných struktur) je možné získat pomocí makra *<TYP_PERFERIE>_COUNT*.

Položka „id“ udává unikátní číslo periferie.

Dále následují ukazatele na funkce, které může ovladač vyplnit tím, že do ukazatelů zapíše adresy implementovaných funkcí. Implementace všech funkcí (kromě inicializační) je volitelná.

Funkce, které může ovladač implementovat:

- Peripheral_status set (uint8_t index, const char *parameter, const char *value)
- Peripheral_status get (uint8_t index, const char *parameter, char *data, uint32_t size)
- void read (uint8_t index, uint8_t *addr, uint32_t size)
- void write (uint8_t index, uint8_t *addr, uint32_t size)
- void stop (uint8_t index, uint8_t *addr)
- void loop (void)
- void intr (uint8_t index, uint8_t intr)

Všechny funkce mají parametr „index“, který rozlišuje jednotlivé instance jednoho druhu periferie. Implementované funkce by měly být deklarovány jako *static*, aby se předešlo kolizím s ostatními ovladači.

Funkce „set“ a „get“ slouží pro nastavení parametrů periferie. To může být např. přenosová rychlost nebo tímto způsobem může být zpřístupněna registrová mapa periferie. Pro co nejširší kompatibilitu jsou parametr a hodnota předány jako ukazatele na řetězce. Za zpracování (např. převod na čísla) je zodpovědný ovladač.

Funkce o výsledku zpracování informuje zbytek firmwaru pomocí návratové hodnoty. Povolené hodnoty jsou:

- OK – zpracování proběhlo v pořádku,
- INVALID_PARAM – špatný parametr,
- INVALID_VALUE – neplatná hodnota,
- ERROR – došlo k jiné chybě.

Další skupinou jsou funkce „read“, „write“ a „stop“. Funkce „write“ slouží pro zápis většího množství dat do periferie. Parametrem je adresa a velikost bufferu, obsahujícího data určená pro zápis. Funkce „read“ je obdobná s tím rozdílem, že buffer je určen pro data přečtená z periferie.

Po dokončení přenosu musí ovladač zavolat jednu z odpovídajících funkcí:

- void write_finish(uint8_t *base_addr)
  Informuje o dokončení zápisu, parametrem je adresa bufferu předaná pomocí funkce „write“.

- void read_finish(uint8_t *base_addr, uint32_t length)
  Informuje o dokončení čtení, parametrem je adresa bufferu předaná pomocí funkce „read“ a počet přečtených bajtů. Funkce by měla být volána vždy, když dojde k zápisu do bufferu.

Funkce „stop“ slouží ke zrušení přenosu. Ovladač musí přerušit jakoukoliv práci s bufferem. Parametrem funkce je počáteční adresa bufferu, která byla předána pomocí funkcí „read“ nebo „write“.

Funkce „loop“ je firmwarem periodicky volána (hlavní programová smyčka). Ovladač se v této funkci může dotazovat na stav periferií, provádět zápisy, čtení apod.

Poslední funkcí je „intr“, tu musí ovladač implementovat, pokud používá přerušení. Jestliže dojde k přerušení, firmware vyhodnotí jeho zdroj a zavolá funkci „intr“ konkrétního ovladače. Parametrem je číslo instance periferie a číslo přerušení.

Pokud periferie umožňuje použití DMA, může ovladač naplánovat nebo zrušit přenos pomocí funkcí:

- uint8_t dma_request_transfer (Dma_config *config)
  Naplánuje DMA přenos.

- uint8_t dma_cancel_transfer (Dma_config *config)
  Zruší DMA přenos.

Funkce jsou dostupné v hlavičkovém souboru *dmac.h*.

Parametrem obou funkcí je ukazatel s konfigurační strukturou pro DMA. Ovladač by měl testovat návratovou hodnotu. Pokud funkce vrátí hodnotu *DMA_BUSY*, znamená to, že řadič DMA je v současnosti zaneprázdněn a není možné naplánovat přenos.

```
1  struct Dma_config{
2    uint32_t src;
3    uint32_t dst;
4    uint32_t size;
5    Dmac_TR_WIDTH src_width;
6    Dmac_TR_WIDTH dst_width;
7    Dmac_BURST_SIZE src_burst_size;
8    Dmac_BURST_SIZE dst_burst_size;
9    Dmac_INC src_increment;
10   Dmac_INC dst_increment;
11   uint8_t id;
12   uint8_t src_handshake;
13   uint8_t dst_handshake;
14
15   void (*callback)(Dma_config *config);
16   uint32_t transfered;
17 };
```

Zdrojový kód 2.4: Konfigurační struktura pro DMA

Struktura obsahuje položky:

- dst – zdrojová adresa,
- dst – cílová adresa,
- size – počet bajtů, které se mají přenést,
- src_width – určuje šířku transakcí zdroje (bajty, půlslova, slova),
- dst_width – určuje šířku transakcí cíle,
- src_burst_size – velikost bloků zdroje (1, 2, 4, 8, 16, 32, 64 nebo 128 B),
- dst_burst_size – velikost bloků cíle,
- src_increment – určuje, zda se má zdrojová adresa inkrementovat,
- dst_increment – určuje, zda se má cílová adresa inkrementovat,
- id – id periferie (součástí konfigurační struktury ovladače periferie),
- src_handshake – číslo DMA kanálu, pokud se jedná o čtení z periferie, jinak nula,
- dst_handshake – číslo DMA kanálu, pokud se jedná o zápis do periferie, jinak nula,
- callback – volitelný ukazatel na funkci, která se zavolá po dokončení přenosu,
- transfered – počet dosud přenesených bajtů.

2.3.2 Příkazy z PC

Všechny příkazy jsou zakončeny znakem pro nový řádek. Na každý příkaz pošle SoC zpět odpověď.

2.3.2.1 Datové typy

V příkazech se používají tyto datové typy:

- ČÍSLO,
- ŘETĚZEC,
- DATA.

ČÍSLO je možné zapsat buď v desítkové, nebo šestnáctkové soustavě. Pro rozlišení číselných soustav musí čísla v šestnáctkové soustavě začínat dvojicí znaků „0x“ nebo „0X“ (obdobně jako v jazyce C). Pro zápis čísel v šestnáctkové soustavě je možné použít jak malá, tak velká písmena. Rozsah čísla je od nuly až do 2^{32}-1.

Příklady:

- desítkový formát: 1654,
- šestnáctkový formát: 0xA – číslo 10,
- šestnáctkový formát: 0X567 – číslo 1383.

Jsou dvě možnosti, jak může být ŘETĚZEC zapsán. Buď začíná písmenem a následující znaky jsou písmena, čísla a znak „_“, nebo je zapsán v uvozovkách a poté může obsahovat jakýkoliv tisknutelný znak (escapování znaků není dovoleno).

Příklady:

- baud_rate – správně
- "5651"– správně
- 5651 – špatně, jedná se o číslo
- 5x – špatně

DATA začínají znakem pravé ostré závorky („>“), kterou následují bajty vyjádřené v šestnáctkové soustavě (číslice „0“ až „9“, malá a velká písmena „a“ až „f“). Cifry musí vždy vyjadřovat celé bajty, tedy počet cifer musí být vždy sudý.

Příklady:

- >FFCCAA55 – správně
- >05 – správně
- >5 – špatně, cifer není sudý počet

2.3.2.2 Popis příkazů

Firmware implementuje čtyři skupiny příkazů:

1. Příkazy pro práci s buffery
2. Příkazy pro přenosy z/do periferie
3. Příkazy pro konfiguraci periferie
4. Příkazy pro konfiguraci SoC

Příkazy pro práci s buffery

- WRITE <ŘETĚZEC jméno_bufferu> <ČÍSLO posun> <DATA data> - Zapíše data do bufferu na adresu určenou parametrem „posun“.
- READ <ŘETĚZEC jméno_bufferu> <ČÍSLO posun> <ČÍSLO velikost> - Přečte „velikost“ bajtů dat z bufferu od adresy „posun“.
- STATUS <ŘETĚZEC jméno_bufferu> - Dotáže se na stav bufferu.

Návratové hodnoty:

- OK - Operace proběhla úspěšně (příkaz WRITE).
- OK <DATA> - Operace proběhla úspěšně, následují data (příkaz READ).
- ERROR unknown buffer - Neznámé jméno bufferu.
- ERROR buffer overflow - Přeplnění bufferu, došlo ke ztrátě dat (příkaz WRITE).
- ERROR invalid offset - Přístup k neplatné adrese v bufferu (příkaz READ).
- ERROR buffer locked - Přístup k bufferu je vyhrazen pro periferii a nelze s ním pracovat (příkazy READ a WRITE).
- STATUS <ŘETĚZEC jméno_bufferu> locked <ČÍSLO počet> - Přístup k bufferu je vyhrazen pro periferii a je v něm uloženo „počet“ bajtů dat (příkaz STATUS).
- STATUS <ŘETĚZEC jméno_bufferu> ready <ČÍSLO počet> - Buffer je k dispozici a je v něm uloženo „počet“ bajtů dat (příkaz STATUS).

Příkazy pro přenosy z/do periferie

- SET <ŘETĚZEC jméno_bufferu> TO <ŘETĚZEC jméno_periferie> - Zahájí přenos z bufferu do periferie.
- SET <ŘETĚZEC jméno_bufferu> FROM <ŘETĚZEC jméno_periferie> - Zahájí přenos z periferie do bufferu.
- SET <ŘETĚZEC jméno_bufferu> STOP - Odebere buffer periferii, přenos je přerušen.

Návratové hodnoty:

- OK - Operace proběhla úspěšně.
- ERROR unknown buffer - Neznámé jméno bufferu.
- ERROR unknown peripheral - Neznámá periferie.
- ERROR not supported - Ovladač periferie tento příkaz nepodporuje.

Příkazy pro konfiguraci periferie

- SET <ŘETĚZEC jméno_periferie> <ŘETĚZEC parametr> <ŘETĚZEC hodnota> - Nastaví parametr periferie na určenou hodnotu.
- GET <ŘETĚZEC jméno_periferie> <ŘETĚZEC parametr> - Přečte hodnotu parametru z periferie.

Návratové hodnoty:

- OK - Operace proběhla úspěšně (příkaz SET).
- OK <STRING value> - Vrací hodnotu požadovaného parametru (příkaz GET).
- ERROR unknown peripheral - Neznámá periferie.
- ERROR invalid parameterd - Neznámý parametr.
- ERROR unknown error - Neznámá chyba v ovladači periferie.
- ERROR not supported - Ovladač periferie tento příkaz nepodporuje.

Příkazy pro konfiguraci SoC

- SET <ŘETĚZEC parametr> <ŘETĚZEC hodnota> - Nastaví parametr na určenou hodnotu
- GET <ŘETĚZEC parametr> - Přečte hodnotu parametru.

Návratové hodnoty:

- OK - Operace proběhla úspěšně (příkaz SET).
- OK <ŘETĚZEC hodnota> - Vrací hodnotu požadovaného parametru (příkaz GET).

2.3.2.3 Zpracování příkazů

Pro zpracování příkazů využívá firmware lexikální a syntaktickou analýzu. Pro vytvoření lexikálního a syntaktického analyzátoru byly použity nástroje *Flex* [26] a *Bison* [27].

Příkaz se nejprve celý uloží do bufferu (konec příkazu je indikován znakem pro nový řádek) a poté se předá lexikálnímu analyzátoru, který ho rozloží na jednotlivé „tokeny“ (klíčová slova, čísla, řetězce). Ty se pak předávají syntaktickému analyzátoru.

Flex generuje analyzátor ve formě zdrojového kódu jazyka C, který nemá žádné závislosti. Je tak možné ho použít i v mikrořadičích bez operačního systému. Analyzátor je realizován formou deterministického konečného automatu a dokáže zpracovat vstupní příkaz se složitostí $O(n)$. Jedinou nevýhodou je, že vytvořený analyzátor potřebuje podporu pro dynamicky alokovanou paměť.

```
ws              [ \t]
num             [0-9]
hnum            ({num}|[a-f]|[A-F])
string          ([a-z]|[A-Z]|_)({num}|[a-z]|[A-Z]|_)*

%%
WRITE{ws}           return WRITE;
READ{ws}            return READ;
STATUS{ws}          return STATUS;
SET{ws}             return SET;
GET{ws}             return GET;
FROM{ws}            return FROM;
TO{ws}              return TO;
STOP                return STOP;

{ws}+       ;
{num}+      {yylval.num = strtol(yytext,NULL,10); return NUMBER;}
0(x|X){hnum}{1,8} {yylval.num = strtol(yytext,NULL,16); return
    NUMBER;}
>({hnum}{hnum})*  {yylval.str.data = yytext+1; yylval.str.length =
     yyleng -1; return DATA;}

{string}           {yylval.str.data = yytext; yylval.str.length =
     yyleng; return STRING;}
[\"][^"]*[\"]      {yylval.str.data = yytext+1; yylval.str.length =
     yyleng-2; return STRING;}
```

Zdrojový kód 2.5: Ukázka pravidel lexikálního analyzátoru

Pro popis „tokenů“ jsou využity regulární výrazy. Klíčová slova a řetězce jsou rozpoznány, bílé znaky ignorovány, číslice jsou převáděna na čísla.

Po provedené lexikální analýze přichází na řadu syntaktický analyzátor vytvořený pomocí nástroje *Bison*. Analyzátor přijímá posloupnost „tokenů“ a na základě gramatiky určuje, o jaký druh příkazu se jedná. Když rozpozná příkaz, extrahuje z „tokenů“ data a zavolá odpovídající funkci firmwaru.

```
start:
WRITE STRING NUMBER DATA END{
$2.data[$2.length] = '\0';
parser_write($2.data, $3, $4.data);}|
...
GET get;
get:
STRING END{
$1.data[$1.length] = '\0';
parser_get_var($1.data);}|

STRING STRING END{
$1.data[$1.length] = '\0';
$2.data[$2.length] = '\0';
parser_get_per($1.data, $2.data);
};
```

Zdrojový kód 2.6: Ukázka gramatiky syntaktického analyzátoru

Pokud posloupnost „tokenů“ neodpovídá žádnému příkazu, zavolá analyzátor funkci *parser_error*.

Díky použití nástrojů *Flex* a *Bison* lze efektivně zpracovávat přijaté příkazy. Rovněž je možné sadu příkazů kdykoliv v budoucnu snadno upravit nebo rozšířit.

2.3.3 Linker skript

Linker skript říká překladači, jaké v systému existují paměťové sekce a jaké části programu do nich má umístit. Při překladech programů pro běžné mikrokontroléry stačí obvykle použít linker skript dodávaný spolu s překladačem. Vytvořený SoC však nemá sekce pevně dané (mění se např. velikost RAM a ROM), proto bylo potřeba vytvořit skript vlastní. Pomocí linker skriptu lze umístit určité proměnné na konkrétní místo paměti, čehož je využito pro komunikaci s *testbenchem*.

Skript definuje dvě paměťové oblasti, *rom* pro program a *sram* pro data. Do oblasti *rom* vloží na začátek sekci *vectors*, ve které jsou uloženy vektory přerušení, dále pak vloží sekci *text* obsahující samotný program. Na začátek oblasti *sram* jsou vloženy sekce *data* (inicializované globální a statické proměnné). Protože však paměť RAM není inicializovaná, není možné data do této paměti uložit hned. Kopie dat musí být uložena v paměti ROM a do RAM se nahraje až po spuštění programu. Uložení kopie dat do oblasti *rom* se zajistí pomocí přiřazení „>sram AT >rom“. Dále se do oblasti *sram* přiřadí sekce *bss* (globální proměnné), které se musí podle standardu C po spuštění programu nastavit na nulu. Další sekcí je *heap* (halda). Jedná se o prostor používaný pro dynamickou alokaci paměti. Na konec paměti je umístěn zásobník, který roste směrem dolů proti sekci *heap*.

```
MEMORY
{
  rom    :    ORIGIN = 0x00000000, LENGTH = 32k
  sram     :    ORIGIN = 0x20000000, LENGTH = 8k
}
_heap_size = 2K;
_end_stack = ORIGIN(sram) + LENGTH(sram);

SECTIONS {
  . = ORIGIN(rom);
  .vectors :
  {
    KEEP (*(.vectors))
  } >rom
  .text :
  {
    *(.text*)
    *(.rodata)
    *(.rodata*)
    _end_text = .;
  } >rom
  .data :
  {
    _start_data = .;
    *(.testbench)
    *(.data)
    *(.data*)
    _end_data = .;
  } >sram AT >rom

  . = ALIGN(4);

  .bss :
  {
    _start_bss = .;
    *(.bss)
    *(.bss*)
    _end_bss = .;
  } >sram

  . = ALIGN(4);

  .heap :
  {
    _heap_start = .;
    . = . + _heap_size;
    _heap_end = .;
  } > sram

  . = ALIGN(4);
}
_end = .;
PROVIDE(end = .);
```

Zdrojový kód 2.7: Ukázka linker skriptu

Sekce *data* obsahuje jako první podsekci *testbench*. V této sekci je umístěna proměnná, která slouží pro komunikaci programu s *testbenchem* při simulaci. Díky linker skriptu je vždy zaručeno, že tato proměnná bude umístěna vždy na stejném místě, a to na začátku paměti.

Linker skript také může definovat symboly, které jsou přístupné z jazyka C. V tomto případě začátky a konce všech sekcí. Symboly nemají žádnou hodnotu, mají pouze adresu, proto se k nim v jazyce C musí přistupovat pomocí operátoru reference (&).

2.3.4 Inicializace procesoru

Před startem funkce *main* je obvykle potřeba provést několik kroků:

1. Nastavit ukazatel na zásobník, aby bylo možné volat podprogramy a funkce (resp. se z nich vracet).
2. Zkopírovat sekci *data* z ROM do RAM.
3. Vynulovat sekci *bss*.
4. Zavolat funkci *main*.

Tyto kroky obvykle vyžadují použití assembleru, procesor *Cortex-M0* je však možné inicializovat pouze s použitím jazyka C.

Ukazatel na zásobník není potřeba nastavovat, nastavení provádí sám procesor. Adresu zásobníku si načte z paměti ROM z adresy nula (0x00). Na další adrese (0x04) se pak nachází adresa funkce, od které začne procesor provádět program. Na následujících adresách je tabulka přerušení.

Potřebné hodnoty jsou uloženy v poli ukazatelů. Linker skript automaticky nastavuje ukazatel na zásobník *_end_stack*. Aby bylo pole umístěno na začátku paměti ROM, je nutné ho umístit do zvláštní sekce *vectors* pomocí atributu pro překladač (*section*). Linker skript pak sekci *vectors* umístí vždy na začátek paměti ROM.

```
1  void *vector_table[] __attribute__ ((section(".vectors"))) = {
2    &_end_stack,
3    Reset_Handler,
4    NMI_Handler,
5    HardFault_Handler,
6    ...
7    UART_interrupt,
8    dma_interrupt,
9    IRQ2,
10   IRQ3,
11   ...
12   IRQ31
13 };
```

Zdrojový kód 2.8: Tabulka přerušení

Dále je potřeba zkopírovat sekci *data*, která obsahuje inicializované globální a statické proměnné. Data jsou v paměti umístěna za koncem programu (ukazatel *_end_text*) a kopírují se na začátek paměti RAM (ukazatel *_start_data*). Kopírování probíhá až k ukazateli *_end_data*.

Sekce *bss* obsahuje neinicializované globální a statické proměnné a stačí ji vynulovat. Sekce je ohraničena ukazateli *_start_bss* a *_end_bss*.

Po inicializaci paměti je možné zavolat funkci *main*.

```
1  void Reset_Handler(void) {
2  uint8_t *src, *dst;
3
4  /* Copy data section from RAM to RAM */
5  src = (uint8_t *)&_end_text;
6  dst = (uint8_t *)&_start_data;
7  while (dst < (uint8_t *)&_end_data)
8  *dst++ = *src++;
9
10 /* Clear the bss section */
11 dst = (uint8_t *)&_start_bss;
12 while (dst < (uint8_t *)&_end_bss)
13 *dst++ = 0;
14
15 main();
16 }
```

Zdrojový kód 2.9: Inicializace paměti

2.3.5 Dynamicky alokovaná paměť

O dynamickou alokaci se stará standardní knihovna jazyka C. Aby bylo možné používat dynamicky alokovanou paměť (funkce *malloc*, *realloc* a *free*), musí tato knihovna vědět, jakou část paměti může pro alokace použít. K tomu stačí implementovat funkci *_sbrk*, jejíž parametrem je počet bajtů, který určuje o kolik chce standardní knihovna posunout hranici haldy. Pokud je parametr kladný, halda se zvětšuje, pokud záporný, halda se zmenšuje.

Funkce vrací ukazatel na původní hranici haldy. Tedy na začátek nově alokované části paměti.

Při nedostatku paměti může funkce alokaci odmítnout. V takovém případě nastaví globální proměnnou *errno* na hodnotu *ENOMEM*.

Na začátku je velikost haldy nula bajtů a její začátek je nastaven na adresu *_heap_start*, která je nastavena pomocí linker skriptu.

```
1  void *_sbrk (int incr)
2  {
3    static uint8_t *heap_barrier = &_heap_start;
4    void *prev_heap_barrier = heap_barrier;
5
6    if(heap_barrier + incr > &_heap_end)
7    {
8      errno = ENOMEM;
9      return (void*) -1;
10   }
11
12   heap_barrier += incr;
13   return prev_heap_barrier;
14 }
```

Zdrojový kód 2.10: Funkce _sbrk potřebná pro dynamicky alokovanou paměť

2.3.6 Překlad

Pro překlad firmwaru je použit překladač *GNU ARM Embedded Toolchain* [28] dostupný ze stránek firmy *ARM*. Dále jsou pro překlad potřeba nástroje *Flex* [26] a *Bison* [27].

Součástí zdrojových kódů firmwaru je *Makefile*, pro překlad tedy stačí spustit příkaz *make*. Vytvořený binární soubor je automaticky převeden do formy, kterou je možné použít pro inicializaci paměti ve *Verilogu* (viz kapitola 2.2.3.2).

2.4 Generátor SoC

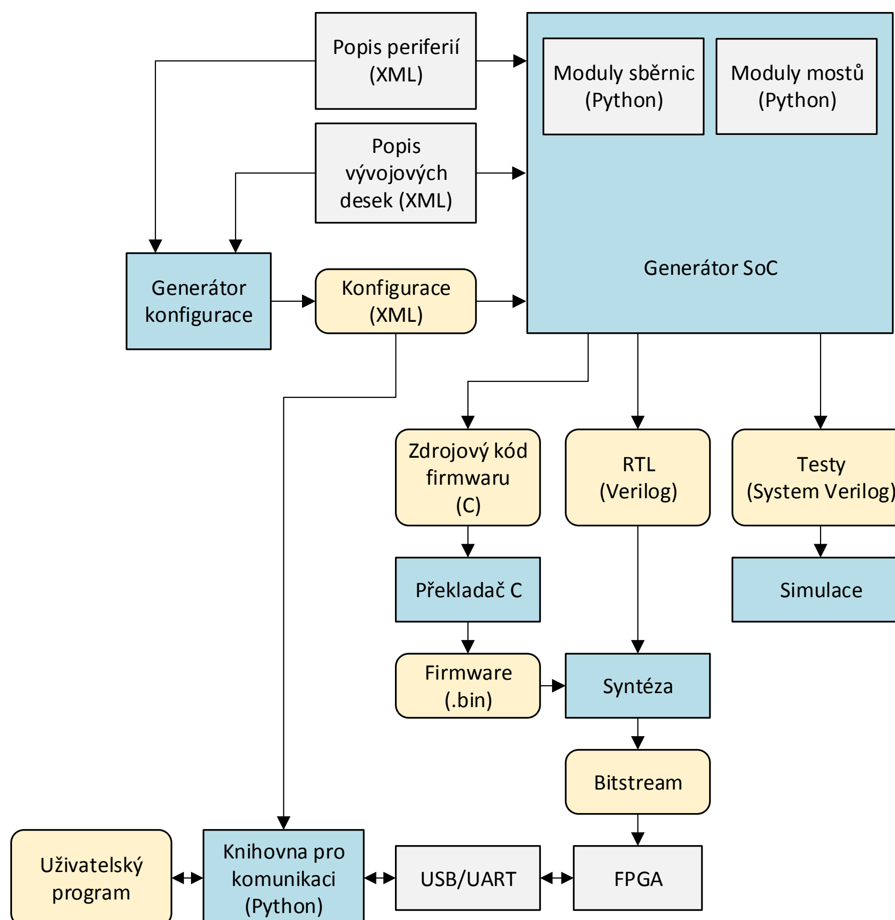
Výstup generátoru se skládá z několika částí.

- Zdrojových kódů SoC (Verilog-2001),
- testbenchů pro SoC a přidané periferie (System Verilog),
- zdrojových kódů firmwaru (jazyk C).

Pro vytvoření RTL části SoC jsou použity moduly pro sběrnice, mosty (angl. *bridge*). Tyto moduly je možné do systému přidávat a rozšiřovat tak jeho možnosti.

2.4.1 Proces vytváření SoC

Proces generování SoC tvoří více kroků. Nejprve je potřeba načíst *XML* soubor, který obsahuje konfiguraci požadovaného SoC (kap. 2.4.2). Poté se načtou všechny moduly sběrnic a periferií. Z těchto modulů se sestaví hierarchické propojení všech sběrnic a mostů. Nejvyšší sběrnice v hierarchii je *AHB-Lite* (sběr-



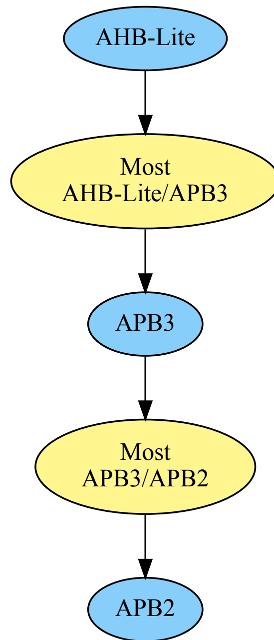
Obrázek 2.13: Proces vytváření SoC

nice procesoru). V současnosti podporuje systém sběrnice *AHB-Lite*, *APB3* a *APB2* (obr. 2.14).

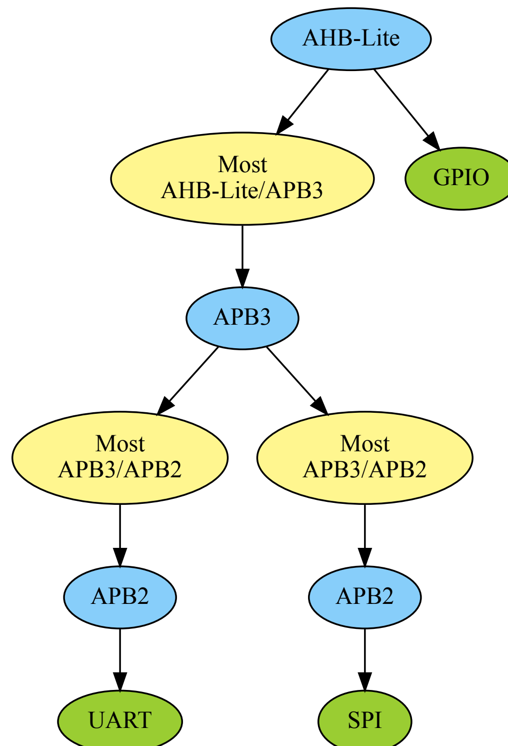
Následně se požadované periferie připojují k daným sběrnicím v hierarchii. Pokud je potřeba ke sběrnici připojit více periferií nebo mostů a daná sběrnice to nepodporuje, je periferie či most připojen ke sběrnici, která je v hierarchii výše (jako např *APB2* v obr 2.15). Na závěr jsou z hierarchie odstraněny sběrnice a mosty, ke kterým není připojena žádná periferie.

Všechny periferie musí být v paměťovém prostoru umístěny tak, aby počáteční adresy periferií byly v paměti zarovnány na velikost dané periferie.

Pro periferie je vyhrazeno až 512 MB paměťového prostoru, do kterých systém automaticky namapuje vytvořené periferie. Periferie a mosty na stejné sběrnici mapuje hned za sebou, aby se předešlo plýtvání adresním prostorem.



Obrázek 2.14: Hierarchie sběrnic a mostů



Obrázek 2.15: Hierarchie s připojenými periferiemi GPIO, UART a SPI

Poté, co je každé periferii přidělen paměťový prostor, je možné vygenerovat RTL popis v jazyce *Verilog*.

Následně jsou vytvořeny soubory potřebné pro překlad firmwaru, *testbench* a testy pro všechny periferie.

Na konci procesu vytváření SoC obsahuje adresář s projektem všechny potřebné soubory pro vytvoření SoC. Před implementací v FPGA je ještě potřeba provést překlad firmwaru (viz kapitola 2.3.6).

2.4.2 Konfigurace SoC

Konfigurace požadovaného SoC je uložena v souboru pomocí textového formátu *XML*, který umožňuje verzování, archivaci a ruční upravování. Pro větší komfort je ale možné konfiguraci vytvořit pomocí grafického rozhraní (obr. 2.16).

Grafické rozhraní je napsáno v jazyce *Python* a využívá multiplatformní knihovny *Qt*. Dynamicky načítá dostupné periferie a jejich konfiguraci a umožňuje nastavit všechny parametry vytvářeného SoC. Konfiguraci ukládá do souboru a umožňuje její opětovné načtení.

V konfiguraci je potřeba nastavit jméno projektu (od něho je odvozen název vytvořeného adresáře s projektem). Určité parametry jako frekvence a jména pinů jsou automaticky předvyplněny při výběru FPGA desky.

Dále je možné přidávat a odebírat samotné periferie a zapínat u nich podporu pro DMA a přerušení.

Aby bylo možné přenášet data z/do periferie, je nutné přidat alespoň jeden buffer.

2.4.3 Moduly periferií

Modul periferie je tvořen *XML* souborem, který popisuje danou periferii. Všechny periferie jsou načítány dynamicky, tzn. pro přidání nové periferie stačí vytvořit pouze jeden nový soubor.

2.4.3.1 Struktura XML souboru

Soubor má jeden kořenový element „peripheral“, který má následující potomky:

- description – název periferie (pod tímto názvem je zobrazena v grafickém rozhraní konfigurátoru),
- type – identifikátor periferie,
- topmodule – název hlavního modulu ve *Verilogu*,
- filelist – seznam souborů s RTL popisem,

Obrázek 2.16: Grafické rozhraní pro konfiguraci

- driver – sekce ovladače
  - initfunction – jméno inicializační funkce pro ovladače,
  - filelist – seznam souborů s ovladačem (.c, .h),
- testcases – seznam testů pro periferii,
- bus – název sběrnice,
- memorymapsize – velikost potřebného paměťového prostoru periferie,
- ports – seznam portů periferie,
  - name – jméno portu,
  - portname – jméno portu, tak jak je definováno v RTL,
  - direction – směr portu (vstupní (input), výstupní (output), obousměrný (bidir))
- interrupts – volitelná sekce týkající se přerušení,
  - name – jméno přerušení,
  - portname – jméno signálu pro přerušení v RTL,
  - alwaysenabled – volitelný parametr
- DMAs – volitelná sekce týkající se DMA,
  - name – jméno rozhraní DMA,
  - prefix – předpona pro jména signálů pro DMA,
  - alwaysenabled – volitelný parametr.

Použití určitého přerušení nebo DMA lze vynutit elementem „alwaysenabled“. Pokud periferie přerušení nebo DMA vůbec nepodporuje, je možné tyto sekce úplně vynechat. Popis periferie *UART* zachycuje zdr. kód 2.11.

2.4.3.2 RTL část

Periferie může být implementována v jazyce *Verilog* nebo *VHDL*. Hlavní modul však musí být napsán v jazyce *Verilog 2001* a pro deklaraci je nutné používat *ANSI* konvenci. Deklarace rovněž nesmí obsahovat signály, které nejsou popsány v *XML* souboru. Pokud toto periferie nesplňuje, je potřeba vytvořit obálku (angl. *wrapper*) a přebytečné signály nezapojit nebo nastavit na patřičné hodnoty.

Signály týkající se sběrnice není potřeba pomocí *XML* souboru definovat, stačí aby byly potřebné signály definované v modulu. Systém je dokáže identifikovat a správně zapojit sám. Poradí si přitom s různou velikostí písmen či drobnými odlišnostmi ve jménu, jako jsou předpony a přípony.

Signály pro přerušení musí být aktivní v log. jedničce. Signály pro DMA musí používat rozhraní popsané v sekci 2.2.4.1.

```
<peripheral>
  <description>Synopsys DesignWare UART</description>
  <type>DW_APB_UART</type>
  <topmodule>s3_valkit_uart</topmodule>
  <filelist>
    <file>s3_valkit_uart.v</file>
    <file>../../../3rdparty/synopsys/uart/DW_apb_uart.v</file>
  </filelist>
  <driver>
    <initfunction>DW_apb_uart_init</initfunction>
    <filelist>
      <file>dw_apb_uart.c</file>
      <file>dw_apb_uart.h</file>
    </filelist>
  </driver>
  <testcases>
    <testcase>dw_uart_rw</testcase>
  </testcases>
  <bus>APB2</bus>
  <memorymapsize>1024</memorymapsize>
  <ports>
    <port>
      <name>RX</name>
      <portname>rx_i</portname>
      <direction>input</direction>
    </port>
    <port>
      <name>TX</name>
      <portname>tx_o</portname>
      <direction>output</direction>
    </port>
  </ports>
  <interrupts>
    <interrupt>
      <name>UART interrupt</name>
      <portname>intr_o</portname>
      <alwaysenabled/>
    </interrupt>
  </interrupts>
</peripheral>
```

Zdrojový kód 2.11: Popis modulu UART

2.4.3.3 Ovladač

Rozhraní, které musí ovladač implementovat, je popsáno v kapitole 2.3.1.1. Do *XML* s konfigurací je potřeba zapsat jméno inicializační funkce.

2.4.4 Moduly sběrnic

Nové sběrnice je možné přidat pomocí modulů. Moduly jsou napsány v jazyce *Python*. Jsou implementované jako třídy, které dědí rozhraní od abstraktní třídy. Jeden modul může implementovat více sběrnic. To je výhodné, pokud se určité sběrnice liší jen minimálně (např. různé verze sběrnic *AMBA*).

Modul je zodpovědný za vytvoření propojení mezi masterem a periferiemi, popř. mosty. V nejjednodušším případě se jedná pouze o propojení potřebných signálů. Pokud je ale připojených zařízení více, je nutné, aby modul vytvořil multiplexer. Modul implementuje tyto metody:

- get_supported_buses()
  Vrací seznam sběrnic, který modul podporuje.

- get_master_signals(bus)
  Vrací seznam signálů a jejich šířky, které má master. Vstupním parametrem je jméno sběrnice.

- get_slave_signals(bus)
  Vrací seznam signálů a jejich šířky, které má slave. Vstupním parametrem je jméno sběrnice.

- get_mux_support(bus)
  Vrací hodnotu *True* nebo *False* podle toho, zda daná sběrnice podporuje připojení více zařízení. Vstupním parametrem je jméno sběrnice.

- generate_source(bus, name, master_signal_map, slave_signal_map, memory_map)
  Vrací kousek vytvořeného zdrojového kódu v jazyce *Verilog*. Vstupní parametry určují jméno sběrnice (bus), unikátní identifikátor (name), mapování signály typu master (master_signal_map) a signály typu slave (slave_signal_map) a paměťovou mapu daných zařízení.

- get_filelist()
  Volitelná metoda, která vrací seznam RTL souborů potřebných pro vytvoření mostu.

2.4.5 Moduly mostů sběrnic

Mosty (angl. *bridge*) umožňují převod jedné sběrnice na druhou. Pro zdrojovou sběrnici se most chová jako slave, pro cílovou sběrnici jako master.

Modul implementuje tyto metody:

- get_supported_bridges()
  Vrací dvojici (angl. *tuple*), kterou tvoří zdrojová a cílová sběrnice.

- generate_source(name, from_, from_bus_module, to, to_bus_module, source_signal_map, dest_signal_map)
  Vrací zdrojový kód vytvořeného mostu v jazyce *Verilog*. Vstupní parametry jsou jméno mostu (name), zdrojová a cílová sběrnice (from_ a to), ukazatele na instance sběrnicových modulů (from_bus_module a to_bus_module) a mapování signálů zdroje a cíle (source_signal_map a dest_signal_map).
- get_filelist()
  Volitelná metoda, která vrací seznam RTL souborů potřebných pro vytvoření sběrnice.

2.4.6 Spuštění programu

Generátor je konzolová aplikace, která ke svému běhu vyžaduje *Python* ve verzi 3.5 nebo vyšší.

Jediným povinným parametrem je cesta ke *XML* souboru s konfigurací (parametr -*c*). Volitelnými parametry lze nastavit výstupní adresář (-*o*), zobrazit nápovědu (-*h*) nebo zapnout rozšířený výstup (-*v*).

```
1 $ ./valkit.py -c config.xml
2 Loading buses...
3 Loading bridges...
4 Loading config file...
5 Creating memory map...
6
7 SoC hierarchy:
8 AHB-Lite
9 +---APB3 apb3_0                          offset: 0       size: 4096
10     +---APB2 apb2_0                      offset: 0       size: 1024
11         +---DW_APB_I2C i2c               offset: 0       size: 1024
12     +---APB2 apb2_1                      offset: 1024    size: 1024
13         +---DW_APB_UART uart             offset: 1024    size: 1024
14     +---APB2 apb2_2                      offset: 2048    size: 1024
15         +---SPIM_DW_APB_SSI spim         offset: 2048    size: 1024
16     +---APB2 apb2_3                      offset: 3072    size: 1024
17         +---SPIS_DW_APB_SSI spis         offset: 3072    size: 1024
18 +---GPIO sw                              offset: 4096    size: 4096
19 +---GPIO led                             offset: 8192    size: 4096
20
21
22
23 Generating source...
24 Creating project dir...
25 Creating files...
26 Generating firmware files...
27 Generating testbench...
28 Generating FPGA releated files...
29 All done ;-)
```

Zdrojový kód 2.12: Ukázka spuštění generátoru

2.5 FPGA desky

Pro realizaci byla vybrána dvojice vývojových desek *Nexys Video Artix-7 FPGA* (obr. 2.17) a *DE1-SoC* (obr. 2.18).

2.5.1 Nexys Video Artix-7 FPGA

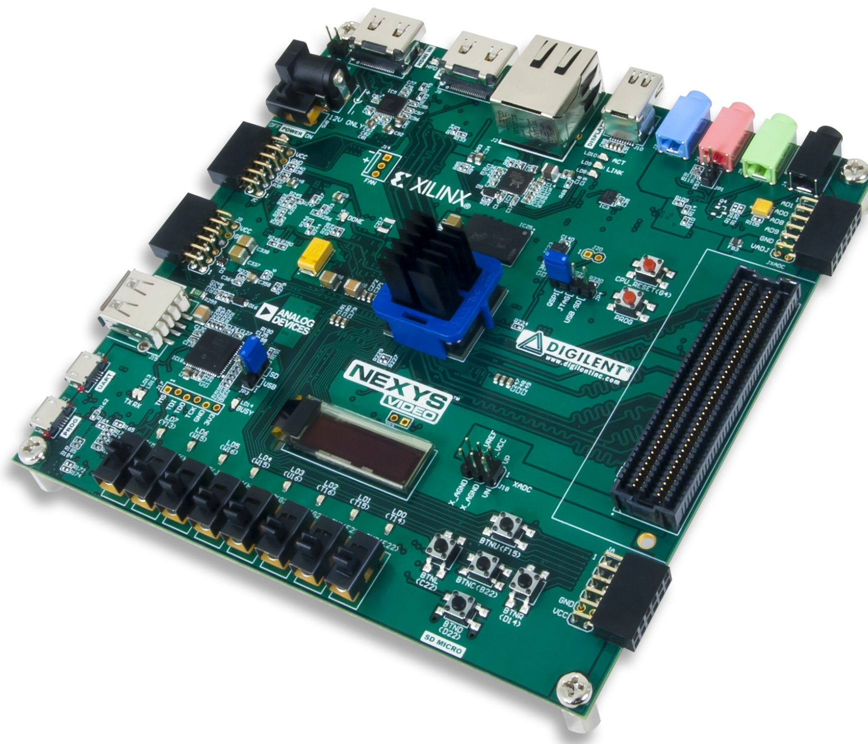
Vlastnosti vývojové desky *Nexys Video Artix-7 FPGA* [29]:

- 33650 logických řezů (angl. *slice*),
- necelých 13 Mbitů rychlých blokových pamětí,
- 8x LED,
- 8 přepínačů a 6 tlačítek,
- A/D převodník (XADC),
- čtyři konektory typu PMOD,
- 160-pinový FMC LPC konektor,
- *USB/UART* převodník,
- 512 MB RAM,
- 2x HDMI (vstup a výstup),
- DisplayPort (výstup),
- monochromatický OLED display (128x32),
- konektor pro MicroSD kartu,
- Audio kodek,
- 10/100/1000 Ethernet,
- USB pro připojení klávesnice nebo myši.

2.5.2 DE1-SoC

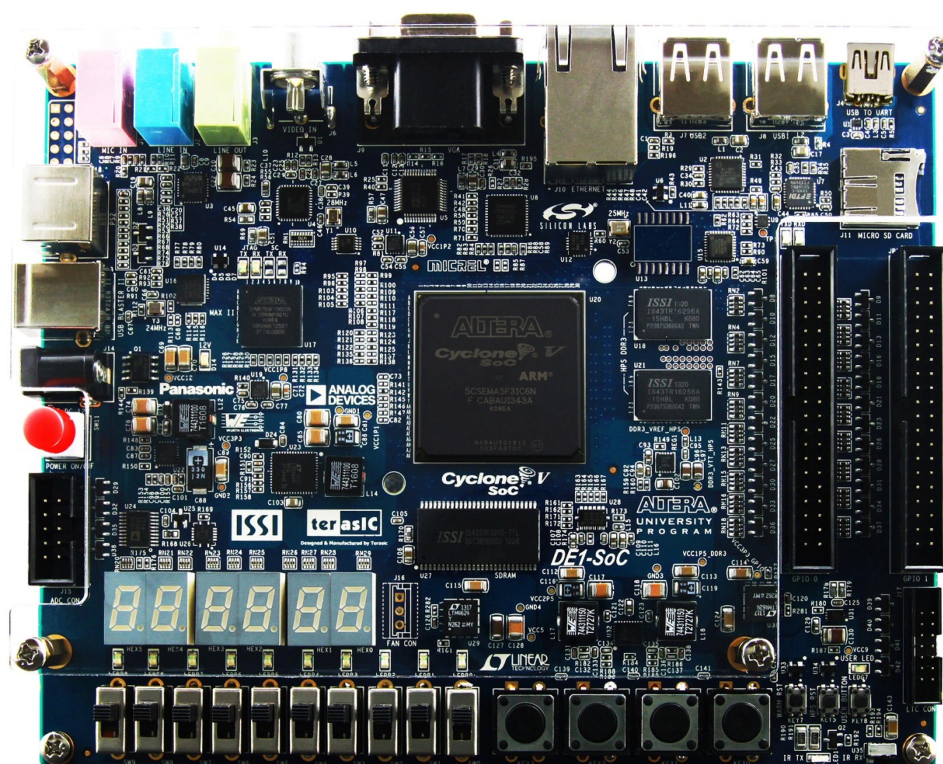
Vlastnosti vývojové desky *DE1-SoC* [30]:

- Dvoujádrový *ARM Cortex-A9*,
- 85 tisíc logických elementů,
- 4450 Kbitů vnitřní paměti,



Obrázek 2.17: Vývojová deska Nexys Video Artix-7 FPGA [29]

- dva čtyřicetipinové konektory,
- *USB/UART* převodník,
- 64MB SDRAM,
- 10/100/1000 Ethernet,
- 11x LED, 6 sedmisegmentových displejů,
- 10 přepínačů a 4 tlačítka
- VGA video výstup



Obrázek 2.18: Vývojová deska DE1-SoC [30]

2.6 Ovládání z PC

Vytvořený SoC používá pro komunikaci rozhraní *UART*. K počítači je připojen pomocí *USB/UART* převodníku, který je integrován na FPGA desce. Používá se standardní rychlost 115200 baudů.

Aby bylo možné SoC ovládat bez speciálního softwaru, je komunikační protokol koncipován jako textový. Pro komunikaci je tak možné použít standardní terminál a posílat potřebné příkazy (popsané v kapitole 2.3.2) ručně nebo využít knihovnu v jazyce *Python*, která komunikační protokol zapouzdřuje.

2.6.1 Knihovna pro komunikaci

Knihovna vyžaduje balíček „PySerial“, který používá k ovládání sériového portu. Knihovna je určená pro *Python 3*.

Veškerou komunikaci obstarává třída „ValkitSerial“. Při vytváření objektu této třídy se konstruktoru předá jméno sériového portu a soubor s konfigurací vytvořený pomocí generátoru konfigurace (kap. 2.4.2).

Třída „ValkitSerial“ poskytuje následující metody:

- write_to_buffer(buffer, data, offset=0)
  Provede zápis do bufferu s názvem „buffer“, „data“ je pole bajtů (angl. *bytearray*), které se mají odeslat, „offset“ je volitelný parametr, který určuje adresu, od které se má zapisovat.

- read_from_buffer(buffer, offset=0, length=-1)
  Provede čtení z bufferu s názvem „buffer“, „offset“ a „length“ jsou volitelné parametry, které určují, od které adresy se má číst a kolik bajtů se má přečíst.

- stop_buffer(buffer)
  Odebere buffer periferii (zruší přenos).

- buffer_status(buffer)
  Zjistí stav bufferu.

- send(peripheral, buffer)
  Odešle obsah bufferu „buffer“ pomocí periferie „peripheral“.

- receive(peripheral, buffer)
  Přijme data z periferie „peripheral“ a uloží je do bufferu „buffer“.

- set_parameter(peripheral, parameter, value)
  Nastaví parametr „parameter“ periferie „peripheral“ na hodnotu „value“.

- get_parameter(peripheral, parameter)
  Přečte parametr „parameter“ periferie „peripheral“.

- set_system_parameter(parameter, value)
  Nastaví parametr „param“ na hodnotu „value“.

- get_system_parameter(parameter)
  Přečte parametr „param“.

V případě potřeby je možné vytvořit více instancí třídy „ValkitSerial“ a ovládat tak více SoC najednou.

```
#!/usr/bin/env python3
import valkitSerial

#Vyvoří objekt pro komunikaci se SoC
s~= valkitSerial.ValkitSerial('/dev/ttyUSB1', '../config.xml')

#Nastaví GPIO led jako výstupní
if not s.set_parameter("led", "direction", "0xff"):
  print (s.last_error)

#Zapíše hodnotu 0xA5 do GPIO led
if not s.set_parameter("led", "data", "0xA5"):
  print (s.last_error)

message = b"Hello from De1-Soc board!\n"
#Zapíše zprávu du bufferu buf1
if not s.write_to_buffer("buf1", message):
  print (s.last_error)

#Vyšle obsah bufferu bu1 pomocí periferie UART
if not s.send("uart", "buf1"):
  print (s.last_error)
```

Zdrojový kód 2.13: Ukázka použití knihovny pro komunikaci

Kapitola 3

Verifikace a validace

V průběhu vytváření SoC byly prováděny simulace, při kterých byly pozorovány průběhy jednotlivých signálů. Rovněž byly vytvořeny testy, které kontrolují funkčnost SoC automaticky.

3.1 Verifikace

Pro ověření správného propojení všech periferií a celkové funkčnosti je možné provést verifikaci pomocí simulace. Součástí vytvořeného SoC jsou dva druhy testů. Systémové, které testují např. funkci procesoru a propojení s počítačem pomocí *UARTu*, a testy periferií, které ověřují chování vygenerovaných periferií.

K verifikaci byl použit simulátor *Incisive* [31] (ve verzi 15.10.013) od firmy *Cadence*.

3.1.1 Systémové testy

Při testování je běžný firmware nahrazen speciálním, který obsahuje testy a provádí verifikaci. Testy jsou tedy psány v jazyce C a běží na procesoru, který je součástí SoC. Test tak může být zaměřen na verifikaci jen určité části hardwaru nebo softwaru.

Test může komunikovat s *testbenchem* pomocí speciální proměnné, která je umístěna vždy na začátku paměti (více v kap. 2.3.3).

3.1.1.1 Komunikace s testbenchem

Test komunikuje s *testbenchem* pomocí funkce *testbench_write*. Pomocí určitého parametru informuje o průběhu verifikace.

Parametrem funkce *testbench_write* může být:

- TESTBENCH_INIT_COMPLETE – firmware dokončil inicializaci,
- TESTBENCH_TEST_OK – test proběhl úspěšně,
- TESTBENCH_TEST_FAIL – test selhal, další testy ale můžou pokračovat,
- TESTBENCH_FINISH – konec verifikace,
- TESTBENCH_ERROR – test selhal, nelze pokračovat v dalších testech, ukončení verifikace.

3.1.2 Testy periferní

Testy jsou zaměřeny na ověření správného propojení mezi periferiemi a procesorem (předpokládá se, že v přidaných modulech sběrnic a mostů (popsaných v kap. 2.4.4 a 2.4.5) mohou být chyby) a základní funkčnosti periferie.

Testování probíhá se standardním firmwarem. Test vytváří stimuly pro periferii a kontroluje odezvy na výstupu periferie. Rovněž simuluje chování PC, které posílá příkazy a přijímá odpovědi pomocí *UARTu*.

Signály, které používá periferie, nemůžou mít vždy stejné jméno (v SoC může být více instancí jedné periferie a docházelo by ke kolizím). Rovněž stimuly ve formě příkazů vypadají pro každý SoC trochu jinak (jiná jména periferií a bufferů). Z těchto důvodů jsou testy generovány dynamicky při vytváření SoC.

Test je tvořen souborem v *(System)Verilogu* doplněném o značky, které jsou při vytváření SoC nahrazeny požadovanou hodnotou. Značky začínají dvojicí symbolů „@@“.

- @@name – značka se nahradí jménem periferie (pro použití v příkazech posílaných z PC),
- @@signal.<jméno_signálu> – značka se nahradí jménem signálu, které odpovídá signálu periferie,
- @@buffer.<velikost_bufferu> – značka se nahradí jménem bufferu požadované velikosti. Při opakovaném použití je značka nahrazena vždy jiným bufferem.

Pokud součástí SoC není buffer požadované velikosti nebo je počet bufferů nedostatečný, test se vůbec nevytvoří.

```
initial begin
gen_reset;
wait_for_cpu;

send_command("SET @@name direction \"0xFFFF\"");
send_command("SET @@name data \"0xAAAA\"");
wait_for_reply("OK");
check_val(@@signal.gpio_io, 16'hAAAA, "GPIO @@name output");

end_simulation;
end
```

Zdrojový kód 3.1: Test zápisu do periferie GPIO

3.2 Validace

Validace proběhla na dvou vývojových deskách FPGA: *Nexys Video Artix-7 FPGA* s FPGA od firmy *Xilinx* a *DE1-SoC* s FPGA od firmy *Altera*.

Pro obě vývojové desky byl vytvořen ukázkový SoC, na kterém probíhala validace. Rovněž byly vytvořeny demonstrační programy v jazyce *Python* využívající knihovnu pro ovládání SoC (kap. 2.6). Oba SoC a demonstrační programy jsou součástí CD.

3.2.1 Design pro FPGA

Předtím, než je možné z vytvořeného SoC vytvořit *bitstream*, je potřeba udělat úpravy specifické pro každou z vývojových desek.

Pokud se vstupní frekvence hodinového signálu liší od požadované, je potřeba přidat fázový závěs (PLL) pro jeho úpravu.

Rovněž je nutné přidat mapování signálů na konkrétní piny (v ideálním případě stačí použít *XDC* soubor dodávaný výrobcem vývojové desky).

Pro vložení všech potřebných souborů do projektu je možné využít vytvořený *Tcl* skript (dostupný jen pro prostředí *Vivado* a *Quartus*).

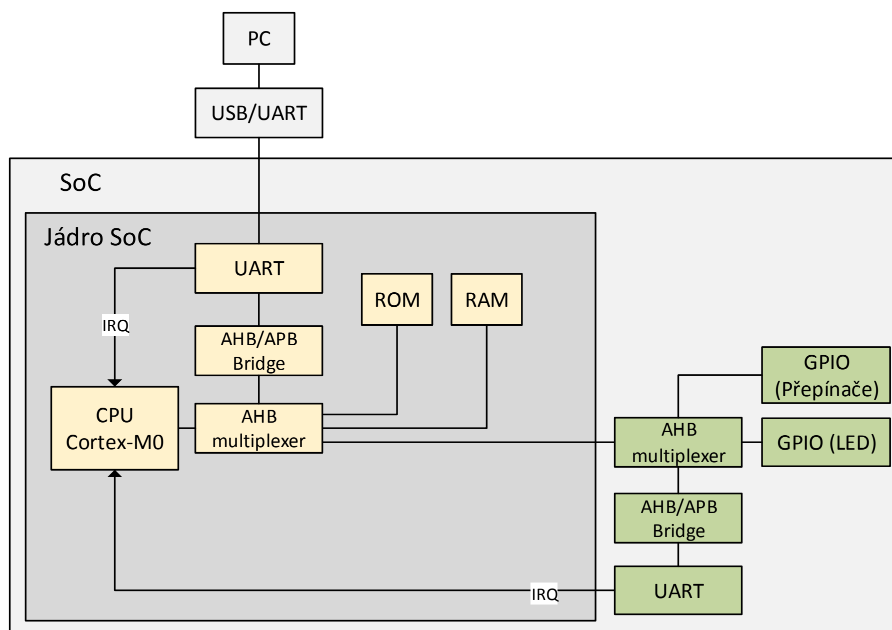
Před spuštěním syntézy je také potřeba provést překlad firmwaru (kap. 2.3.6).

3.2.2 Ukázkový SoC pro Nexys Video Artix-7 FPGA

Ukázkové SoC pro *Nexys Video Artix-7 FPGA* obsahuje rozhraní *UART* a dvojici *GPIO* připojenou k přepínačům a LED diodám (obr. 3.1). Rozhraní *UART* je vyvedeno pomocí konektoru *PMOD JA*. *UART* sloužící k ovládání SoC je připojen k integrovanému *USB/UART* převodníku na desce. Reset je připojen k tlačítku *cpu_resetn*.

Navržený SoC zabírá asi 10 % FPGA (obr. 3.3) a pracuje na frekvenci 50 MHz. Při syntéze bylo nutné nastavit optimalizace pro vyšší frekvenci.

Ukázková aplikace čte stav všech přepínačů a podle toho nastavuje stav odpovídajících LED diod. Přes *UART* je periodicky odesílán řetězec: „Hello



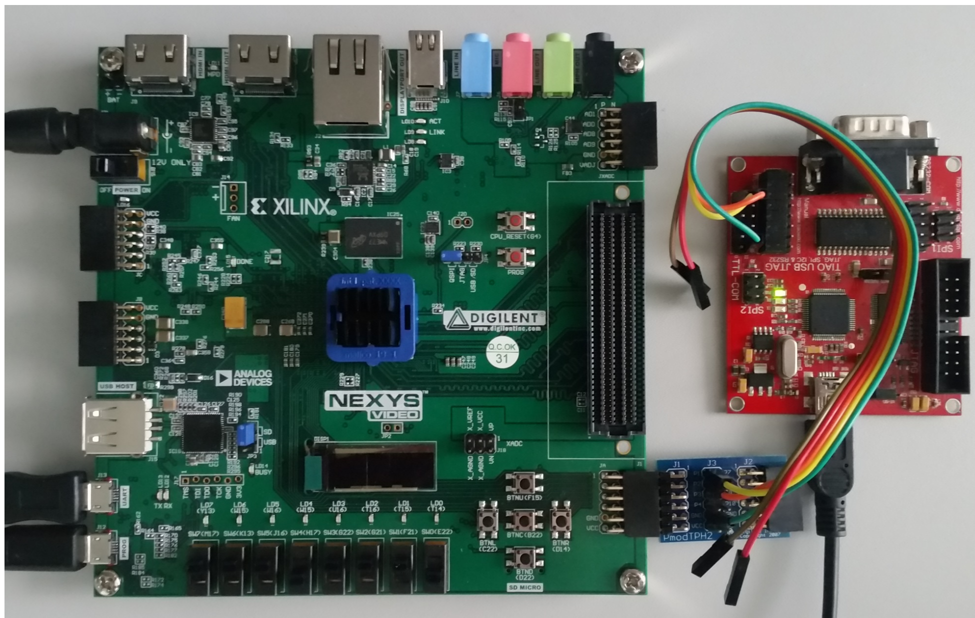
Obrázek 3.1: Ukázkový SoC pro Nexys Video Artix-7 FPGA

Využití řezů	3469/33450 (10.37 %)
Řezy použité pro logiku	1926
Řezy použité pro paměť	1543
Využití LUT	11328/133800 (8.47 %)
LUT jako logika	8256/133800 (6.17 %)
LUT jako paměť	3072/46200 (6.65 %)
Celkem registrů	2544 (0.95 %)
Celkem DSP bloků	3/740 (0.41 %)

Tabulka 3.1: Výsledek implementace pro vývojovou desku Nexys Video Artix-7 FPGA

from Nexys Video Artix-7 FPGA board!“. Přes *UART* je také možné přijímat data. Všechny přijaté znaky se zobrazí na standardním výstupu ukázkové aplikace. *UART* používá rychlost 115 200 baudů.

Na desce bylo úspěšně otestováno chování *GPIO* použitých pro LED a přepínače (nastavení a čtení všech možných hodnot). Rovněž bylo otestováno rozhraní *UART*. Testováno bylo přijímaní a odesílání zpráv a práce s buffery. Pro testování byl použit externí *USB/UART* převodník (*TIAO TUMPA* [32], obr. 3.2).



Obrázek 3.2: Nexys Video Artix-7 FPGA s připojeným UARTem

3.2.3 Ukázkový SoC pro DE1-SoC

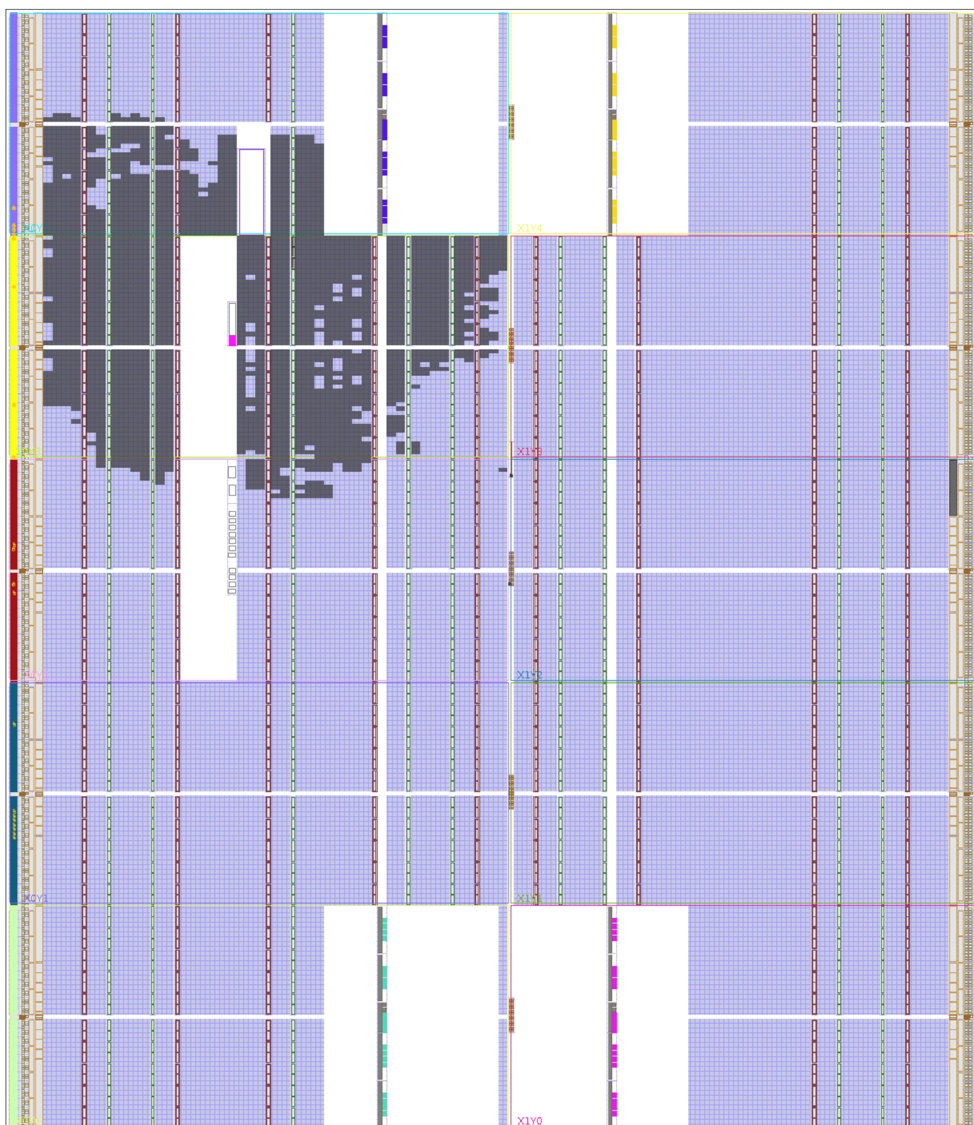
Ukázkové SoC pro *DE1-SoC* obsahuje dvojici *GPIO* připojenou k přepínačům a LED. Dále rozhraní *UART*, *I²C*, dvojici *SPI* (master a slave) (obr. 3.4). Všechna rozhraní jsou vyvedena pomocí konektoru *GPIO 0*. Pro přenos pomocí *SPI* je použit řadič DMA. *UART* sloužící k ovládání SoC je připojen k integrovanému *USB/UART* převodníku na desce. Reset je připojen k tlačítku *KEY 0*.

Využití logiky (ALM)	12,030 / 32,070 (38 %)
Celkem registrů	11030
Celkem paměťových bitů	393,216 / 4,065,280 (10 %)
Celkem RAM bloků	48 / 397 (12 %)
Celkem DSP bloků	2 / 87 (2 %)

Tabulka 3.2: Výsledek implementace pro vývojovou desku DE1-SoC

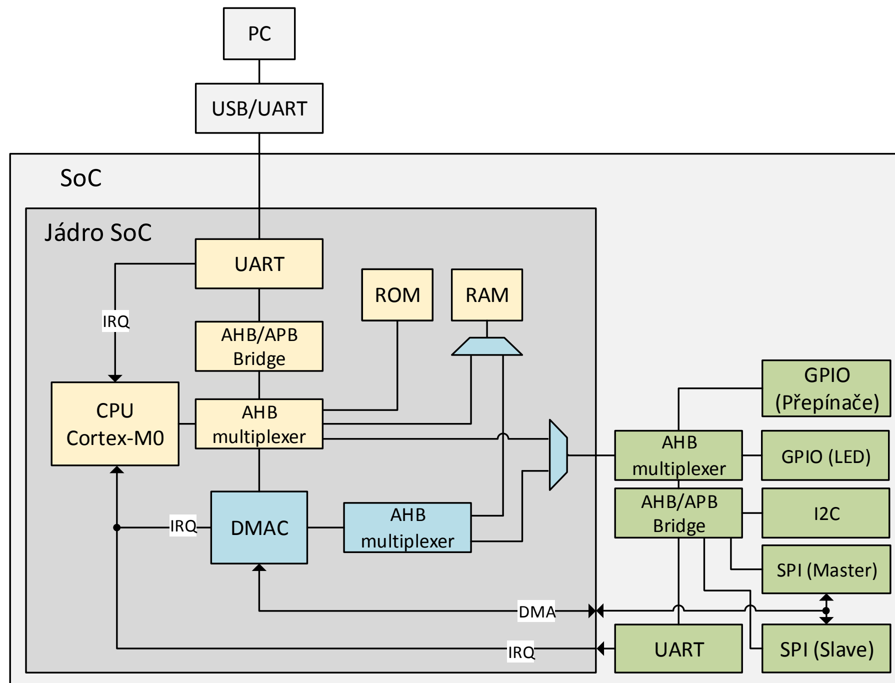
FPGA je zaplněno z necelých 40 % (obr. 3.6) a pracuje na frekvenci 50 MHz (maximální frekvence je 64.52 MHz).

Podobně jako u SoC pro *Nexys Video Artix-7* FPGA čte ukázková aplikace stav všech přepínačů a podle toho nastavuje stav odpovídajících LED diod. Přes *UART* je periodicky odesílán řetězec: „Hello from De1-Soc board!“. Přes *UART* je možné přijímat data. Všechny přijaté znaky se zobrazí na standardním výstupu ukázkové aplikace. *UART* používá rychlost 115 200 baudů.

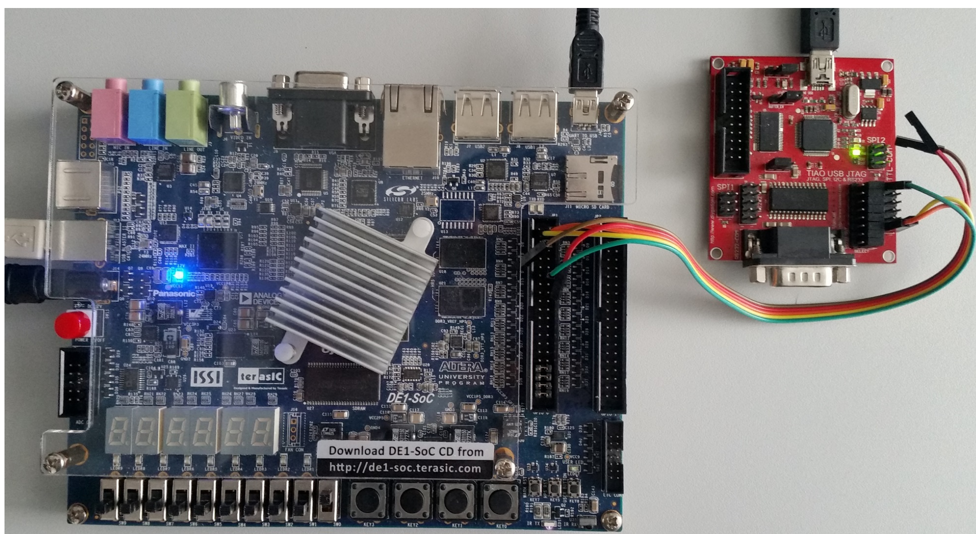


Obrázek 3.3: Rozložení SoC v FPGA na desce Nexys Video Artix-7 FPGA

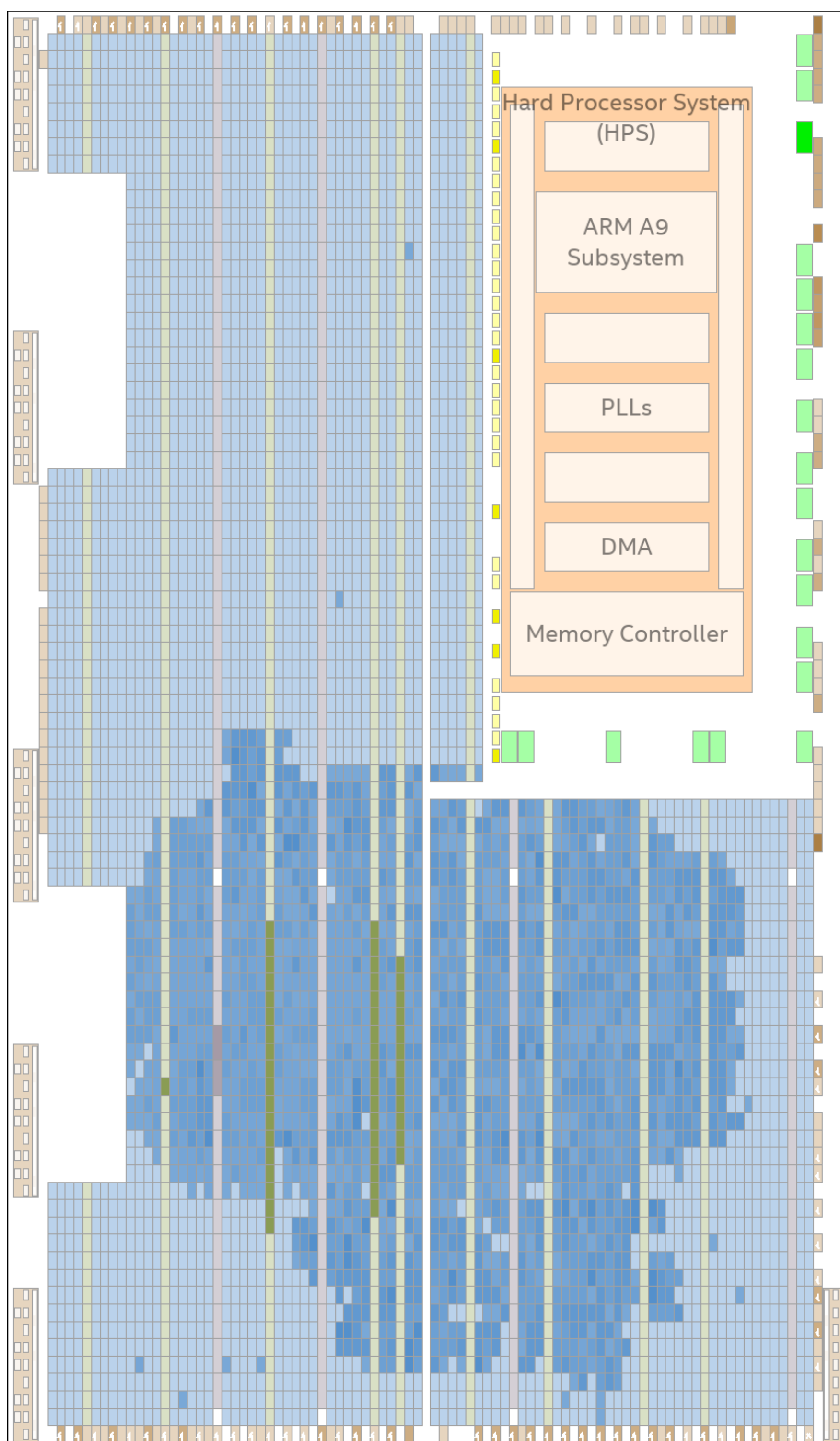
Na desce bylo úspěšně otestováno chování *GPIO* použitých pro LED a přepínače. Rozhraní *UART* bylo testováno pomocí externího převodníku (obr. 3.5). Rovněž byl otestován přenos pomocí *SPI* (komunikace mezi *SPI* master a *SPI* slave, které byly navzájem propojeny). Pro přenosy bylo využito DMA.



Obrázek 3.4: Ukázkový SoC pro DE1-SoC



Obrázek 3.5: DE1-SoC s připojeným UARTem



Obrázek 3.6: Rozložení SoC v FPGA na desce DE1-SoC

Závěr

Práce se zabývala návrhem SoC a nástrojem pro tvorbu systémů na čipu, které je možné ovládat pomocí počítače. Pomocí tohoto nástroje lze vytvořit SoC, obsahující procesor *ARM Cortex-M0* se všemi potřebnými komponentami (paměti, rozhraní *UART* pro komunikaci s PC apod.) a periferie (*GPIO*, *UART*, *SPI*, *I²C*) podle konfigurace uživatele.

Pro konfiguraci bylo vytvořeno grafické rozhraní, které umožňuje pohodlně nastavit všechny parametry požadovaného SoC.

Periferie mohou používat různé sběrnice, nástroj automaticky vytvoří potřebné mosty a propojení. Rovněž v případě potřeby připojí signály pro přerušení a rozhraní DMA.

Nástroj byl koncipován modulárně, což umožňuje jednoduše přidávat podporu pro další periferie a sběrnice. V současnosti nástroj podporuje sběrnice *AHB-Lite*, *APB3* a *APB2* a tyto periferie třetích stran: *GPIO* od firmy *ARM* a *UART*, *SPI* (master i slave) a *I²C* od firmy *Synopsys*.

Správné propojení všech periferií a funkce firmwaru byla ověřena pomocí vytvořeného *testbenche* a testů. Pro simulace byl použit simulátor *Incisive* [31] od firmy *Cadence*.

Práce se rovněž zabývala návrhem univerzálního firmwaru v jazyce C. Pro překlad byl použit překladač *GCC*. Do firmwaru jsou automaticky přidány potřebné ovladače pro implementované periferie. Firmware je tedy vždy sestaven na míru pro konkrétní SoC.

Z PC je tak možné komunikovat se SoC přes *USB* (využívá se *USB/UART* převodník) a ovládat všechny periferie pomocí jednotného rozhraní. Z počítače je možné SoC ovládat pomocí sériového terminálu a textových příkazů nebo lze využít vytvořenou knihovnu v jazyce *Python*.

Vytvořený SoC je určen pro FPGA, není však vázán na konkrétního výrobce. SoC včetně firmwaru a ovládání z PC byl úspěšně otestován na FPGA *Xilinx* (*Artix-7 A200T*) a *Altera* (*Cyclone V*) s posledními verzemi nástrojů *Vivado* [2] a *Quartus* [1].

V rámci práce byly vytvořeny syntetizovatelné modely pamětí RAM a

ROM, most mezi sběrnicemi *AHB-Lite* a *APB3* a arbitr pro sběrnici *AHB-Lite* (vše v jazyce *Verilog*).

Práce využívá procesor od firmy *ARM* a periferie od firmy *Synopsys*. Z licenčních důvodů nejsou tyto komponenty součástí přiloženého CD. SoC lze vytvořit i bez těchto komponent, ale už nepůjde vygenerovat *bitstream* nebo provést simulaci.

Do budoucna by bylo vhodné rozšířit knihovny periférií a sběrnic. Dále přidat podporu pro více druhů procesorů a možnost vytvářet víceprocesorové systémy.

Literatura

[1] Intel Corporation: Intel® Quartus® Prime Design Software [online]. 2017, [cit. 2017-05-02]. Dostupné z: https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html

[2] Xilinx, Inc.: Vivado Design Suite [online]. 2017, [cit. 2017-05-02]. Dostupné z: https://www.xilinx.com/products/design-tools/vivado.html

[3] Xilinx, Inc.: *Vivado Design Suite Tutorial [online]*. [cit. 2017-04-11]. Dostupné z: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug995-vivado-ip-subsystems-tutorial.pdf

[4] Xilinx, Inc.: *MicroBlaze Soft Processor Core [online]*. [cit. 2017-04-11]. Dostupné z: https://www.xilinx.com/products/design-tools/microblaze.html

[5] Intel Corporation: *Nios II Gen2 Processor Reference Guide [online]*. [cit. 2017-04-11]. Dostupné z: https://www.altera.com/documentation/iga1420498949526.html

[6] OPENCORES.ORG and Authors.: *OpenRISC 1000 Architecture Manual [online]*. [cit. 2017-04-11]. Dostupné z: https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.1-rev0.pdf

[7] ARM Limited.: AMBA Specifications [online]. 2017, [cit. 2017-04-11]. Dostupné z: https://www.arm.com/products/system-ip/amba-specifications

[8] ARM Limited.: *AMBA® AXI™ and ACE™ Protocol Specification [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0051a/index.html

[9] ARM Limited.: *ARM® AMBA® 5 AHB Protocol Specification [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0033/index.html

[10] ARM Limited.: *Multi-layer AHB [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.dvi0045b/index.html

[11] ARM Limited.: *AMBA™Specification [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html

[12] ARM Limited.: *AMBA® 3 APB Protocol [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0024b/index.html

[13] Altera Corporation: *Avalon Interface Specifications [online]*. [cit. 2017-04-11]. Dostupné z: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf

[14] OpenCores: *WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores [online]*. [cit. 2017-04-11]. Dostupné z: http://cdn.opencores.org/downloads/wbspec_b4.pdf

[15] Root.cz: *Externí sériové sběrnice SPI a I^{2}C [online]*. [cit. 2017-04-11]. Dostupné z: https://www.root.cz/clanky/externi-seriove-sbernice-spi-a-i2c/

[16] Wikipedia: Serial Peripheral Interface Bus — Wikipedia, The Free Encyclopedia [online]. 2017, [cit. 2017-04-11]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Serial_Peripheral_Interface_Bus&oldid=774698843

[17] NXP Semiconductors: *UM10204 I^{2}C-bus specification and user manual [online]*. Revize v.6, [cit. 2017-04-11]. Dostupné z: www.nxp.com/documents/user_manual/UM10204.pdf

[18] ARM Limited.: *Cortex-M0 Technical Reference Manual [online]*. [cit. 2017-03-14]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf

[19] ARM Limited.: *Cortex-M0 memory map [online]*. [cit. 2017-01-20]. Dostupné z: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0182n/CHDHCADA.html

[20] Altera Corporation: *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide [online]*.

[cit. 2017-04-16]. Dostupné z: https://www.altera.com/en_US/pdfs/literature/ug/ug_ram_rom.pdf

[21] Synopsys, Inc.: DesignWare IP [online]. 2017, [cit. 2017-05-02]. Dostupné z: https://www.synopsys.com/designware-ip.html

[22] Synopsys, Inc.: *DesignWare DW_ahb_dmac Databook*. Revize 2.20a.

[23] Synopsys, Inc.: *DesignWare DW_apb_uart Databook*. Revize 4.00a.

[24] Synopsys, Inc.: *DesignWare DW_apb_ssi Databook*. Revize 4.00a.

[25] Synopsys, Inc.: *DesignWare DW_apb_i2c Databook*. Revize 2.00a.

[26] Estes, W.; Millaway, J.; Paxson, V.; aj.: The Fast Lexical Analyser [online]. 2017, [cit. 2017-04-24]. Dostupné z: https://github.com/westes/flex

[27] Corbett, R.; The GNU Project: GNU Bison [online]. 2017, [cit. 2017-04-24]. Dostupné z: https://www.gnu.org/software/bison

[28] ARM Limited.: GNU ARM Embedded Toolchain [online]. 2017, [cit. 2017-04-24]. Dostupné z: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm

[29] Digilent: Nexys Video Reference Manual [online]. 2017, [cit. 2017-04-11]. Dostupné z: https://reference.digilentinc.com/reference/programmable-logic/nexys-video/reference-manual

[30] Terasic Technologies: *DE1-SoC User Manual [online]*. [cit. 2017-04-11]. Dostupné z: http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=ae336c1d5103cac046279ed1568a8bc3

[31] Cadence Design Systems, Inc.: Incisive Enterprise Simulator [online]. 2017, [cit. 2017-04-28]. Dostupné z: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html

[32] TIAO Corp.: TIAO USB Multi Protocol Adapter User's Manual [online]. 2017, [cit. 2017-05-04]. Dostupné z: http://www.tiaowiki.com/w/TIAO_USB_Multi_Protocol_Adapter_User's_Manual

[33] ARM Limited.: Cortex-M0 Processor DesignStart [online]. 2017, [cit. 2017-05-02]. Dostupné z: https://www.arm.com/products/designstart

Příloha A

Seznam použitých zkratek

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

ANSI American National Standards Institute

APB Advanced Peripheral Bus

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

DMA Direct Memory Access

DMAC Direct Memory Access Controller

DSP Digital Signal Processor

FIFO First In, First Out

FPGA Field Programmable Gate Array

GPIO General Purpose Input/Output

IP Intellectual Property

ISA Instruction Set Architecture

I^2C Inter-Integrated Circuit

LED Light-Emitting Diode

LMB Local Memory Bus

MMU Memory Management Unit

MPU Memory Protection Unit

PC Personal Computer

PLL Phase-Locked Loop

RAM Random Access Memory

RISC Reduced Instruction Set Computing

ROM Read-Only Memory

RTL Register Transfer Level

SoC System on Chip

SPI Serial Peripheral Interface

Tcl Tool Command Language

TWI Two Wire Interface

SCL Serial Clock Line

SDA Serial Data Line

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

XDC Xilinx Design Constraints

XML eXtensible Markup Language

Příloha **B**

Obsah přiloženého CD

Z licenčních důvodů není součástí CD procesor ani periferie. Použitý procesor *ARM Cortex-M0* (verze r1p0-00rel0) lze licencovat pomocí portálu *DesignStart* [33]. Periferie pochází z knihovny *IP DesignWare* [21] (ve verzi 2016.03-SP3).

```
readme.txt ............................ stručný popis obsahu CD
data
  3rdparty ............................ zdrojové kódy třetích stran
  s3 .................................. zdrojové kódy S3 Group
  s3_valkit
    s3_valkit.tar.gz .................. zdrojové kódy implementace
    s3_valkit_DE1_Soc_Demo.tar.gz ..... ukázkový SoC pro De1-SoC
    s3_valkit_Nexys_demo.tar.gz ....... ukázkový SoC pro Nexys Video
                                        Artix-7 FPGA
  thesis .............................. zdrojová forma práce ve formátu LaTeX
text
  DP_Vanc_2017.pdf .................... text práce ve formátu PDF
```