České vysoké učení technické v Praze
Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

Připojení rychlé sériové sběrnice k procesoru PowerPC

Praha, 2010

Autor: Michal Hrouda

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 7.5.2010

Hrouda Michal

podpis

Poděkování

Tímto bych chtěl poděkovat především vedoucímu mé diplomové práce za pomoc při psaní této práce. Velké poděkování patří též firmám Energocentrum Plus s.r.o. a Mikroklima s.r.o. za jejich pomoc a za poskytnutí nezbytných podkladů a vývojových prostředků, bez kterých by tato práce nemohla vzniknout. Děkuji také firmě Sysgo za jejich technickou podporu při řešení vzniklých potíží.

Abstrakt

V této diplomové práci je popsána metoda konfigurace hradlového pole pomocí periferií procesoru PowerPC, jeho připojení k externí sběrnici procesoru. Pro otestování funkčnosti obou předchozích celků byla realizována aplikace v systému PikeOS, která s využitím hradlového pole komunikuje s moduly na sériové sběrnici PIRANHA.

Abstract

This diploma thesis describes method for FPGA configuration by peripheral of PowerPC processor, its connection to processor external memory bus. For tests of previous tasks, the test application in PikeOS system have been written. This application communicates via FPGA with modules connected by PIRANHA serial bus.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Michal Hrouda**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM1 - Řídicí technika

Název tématu: **Připojení rychlé sériové sběrnice k procesoru PowerPC**

Pokyny pro vypracování:

Cílem práce je připojit rychlou sériovou sběrnici PIRANHA k procesoru MPC5200 a integrovat ji do operačního systému PikeOS. Protokol sériové sběrnice je k dispozici v FPGA.

1. Navrhněte a realizujte přímé připojení FPGA k procesoru MPC5200.
2. Napište ovladač do uBoot, který FPGA nakonfiguruje.
3. Navrhněte a realizujte low-level driver pro připojení sběrnice PIRANHA do posixového oddílu operačního systému PikeOS. Při návrhu dbejte zejména na rychlost komunikace, synchronizaci, diagnostiku linky a spolehlivost systému.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Pavel Burget, Ph.D.

Platnost zadání: do konce letního semestru 2010/2011

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 17. 12. 2009

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce se zabývá metodami připojení hradlového pole (dále v práci budu používat již jen zkratku FPGA - Field Programmable Gate Array) k procesoru PowerPC MPC5200, konkrétně realizací připojení na jeho externí paměťovou sběrnici. Toto je popsáno v první části (2) práce. FPGA je programovatelná součástka, je ji tedy nutné po připojení napájení inicializovat, možné metody této inicializace, taktéž nazývané konfigurace, jsou popsány v druhé části (3) této práce. Zde je vybrána jedna konkrétní metoda a popsán postup její realizace v bootloaderu U-Boot. V poslední, třetí, části (4) je vytvořena aplikace na platofrmě PikeOS firmy Sysgo, která provádí komunikaci s moduly na sběrnici Piranha.

Tato práce volně navazuje na moji bakalářskou práci s titulem ‘Operační systémy PikeOS a LINUX‘, přesto pro kompletnost této práce zde budou některé důležité informace zopakovány.

Procesor MPC5200 je představitelem architektury PowerPC, jde o 32bitový procesor, který v sobě integruje jednotku pro operace s plovoucí řádovou čárkou v dvojnásobné přesnosti, jednotku správy paměti, ethernet a další periferie vhodné pro řídící účely.

FPGA jsou moderní polovodičové programovatelné součástky s velmi vysokou hustotou intergace. S jejich pomocí lze realizovat logické kombinační a sekvenční obvody od jednoduchých dekodérů, čítačů přes složitější periferie pro procesorové obvody až po procesory samotné. Spektrum těchto obvodů je stejně tak široké jako jejich využití, jednotlivé typy se mezi sebou liší například počtem vstupně/ výstupních pinů, řádově desítky až stovky, velikostí umístitelného návrhu a dále samozřejmě typem pouzdra. Zatímco u počtu pinů není s porovnáním mezi výrobci žádný spor, ohledně velikosti návrhu už to tak jednoznačné není, výrobce obvykle uvádí počet

bloků v FPGA. Takový blok obvykle obsahuje look-up tabulku (zkráceně LUT) a klopný obvod (zkráceně FF), počet vstupů u LUT má každý výrobce jiný a také počet FF v bloku je mezi výrobci jiný. Díky těmto rozdílům již není možné srovnání na základě počtu bloků, proto výrobci používají definici na základě *ekvivalentních hradel*, kde jejich počet bývá od několika tisíc po řádově miliony. Některé řady FPGA v sobě obsahují i hotové bloky jako například jeden nebo dva procesory PowerPC, akcelerátory pro náročné výpočty, hardwarové násobičky, paměti a další.

PikeOS (4) je systém určený pro embedded zařízení, kde je vyžadována vysoká spolehlivost a robustnost. Nejedná se však o plnohodnotný *operační* systém, jeho hlavní podstatou je virtualizační jádro, které vytváří prostředí pro provoz již plnohodnotných operačních systémů a nebo tzv. nativních úloh. Ve spolupráci s plně konfigurovatelným časovým plánováním tohoto jádra je možné vytvořit úplný realtime systém s vysokou spolehlivostí, neboť časově kritické činnosti jsou vykonávány nativní úlohou a interakce s uživatelem je zajištěna aplikací běžící v prostředí Linuxu.

Kapitola 2

Popis hardware a sběrnice PIRANHA

2.1 Procesor MPC5200B

2.1.1 Popis procesoru MPC5200B

Procesor MPC5200B v sobě integruje vysoce výkonné jádro e300 s rozsáhlým množstvím periferií zaměřených na komunikace a systémové integrace. Jádro e300 je založeno na architektuře jader PowerPC. MPC5200B představuje inovativní I/O subsystém, který odděluje správu periferií od vlastního jádra e300. MPC5200B podporuje architekturu dvojité externí sběrnice. Ta se skládá z rychlé sběrnice pro SDRAM připojené přímo na jádro e300 a sběrnice s názvem LocalPlus bus, která je používána jako obecný interface pro připojení dalších periferních zařízení a ladícího prostředí.

Tento procesor byl vybrán z následujících důvodů

- Velký výpočetní výkon
- Integrována FPU a MMU
- Vhodné periferie pro řídící účely (6x UART, CAN, USB), ethernet řadič
- Oddělená sběrnice pro SDRAM a ostatní periferie
- K dispozici v automotive provedení $\Rightarrow$ spolehlivost, garance dlouhého výrobního cyklu
- Podporován většinou Realtime operačních systému a Linuxem

Základní vlastnosti procesoru MPC5200 lze shrnout do následujících bodů

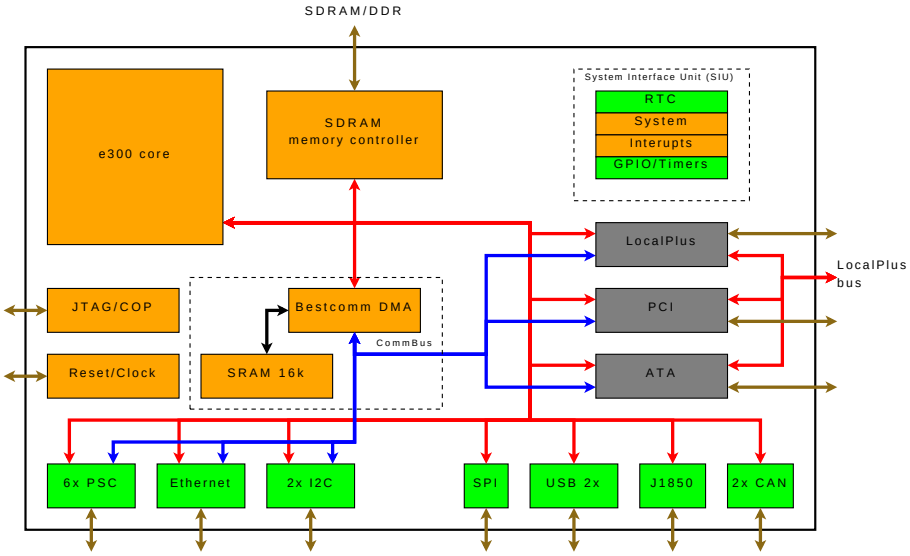
- Jádro e300

    - Superskalární architektura
    - 760 MIPS při 400 MHz (v průmyslovém rozsahu teplot -40°C - 85°C)
    - Jednotka pro operace s plovoucí řádovou čárkou v dvojnásobné přesnosti
    - Jednotka správy paměti
- SDRAM/DDR interface
    - Podpora pracovní frekvence až 132 MHz
    - Podpora režimů SDR a DDR
    - 256 MB adresní prostor na jeden signál Chip select
    - 32bitová šířka datové sběrnice
    - Přímá podpora inicializace a refresh pamětí
- Flexibilní externí sběrnice LocalPlus bus
    - Podpora ROM/Flash/SRAM a dalších paměťově mapovaných zařízení
    - Osm programovatelných signálů Chip select
    - Nemultiplexovaný režim s šířkou dat 8/16/32bit a až 26bit adresy
    - Multiplexovaný režim s šířkou dat 8/16/32bit a až 25bit adresy
- Řadič PCI kompatiblní s verzí 2.2
- Řadič ATA
- Šest programovatelných sériových kontrolérů (PSC)
    - Lze používat v režimu UART/Soft Modem/I2S/AC97/Plně duplexní SPI/IRDA
- Fast ethernet řadič
- Řadič Host USB verze 1.1 (OHCI), k dispozici 2 porty
- Dva řadiče sběrnice I2C
- Řadič SPI
- Dva řadiče sběrnice CAN verze 2.0 A/B
- Ladící rozhraní dle standardu IEEE 1149.1

Na obr. 2.1 je zobrazeno zjednodušené blokové schéma tohoto procesoru. Z blokového schématu je jasně patrná ona dvojitá externí sběrnice, pro SDRAM a LocalPlus bus - ta nás bude zajímat nejvíce.

Řadič SDRAM/DDR je připojen přímo na jádro e300, kdy díky použití na čipu integrované 16kB instrukční a 16kB datové vyrovnávací paměti je dosaženo vysokého

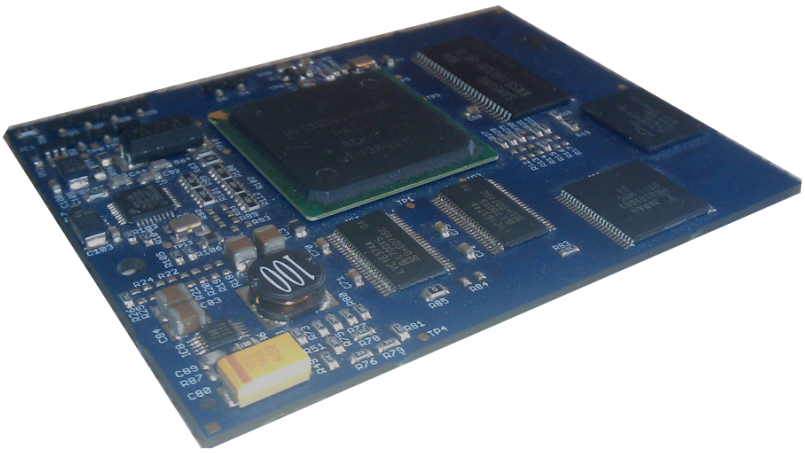
výkonu procesoru pro výpočetně náročné aplikace.
Procesor je dále vybaven integrovaným inteligentním řadičem přímého přístupu do paměti (DMA) BestComm, který umožňuje na jádře nezávislou obsluhu přerušení od periferií, jejich nízkoúrovňovou správu a přesuny bloků paměti.



Obrázek 2.1: Zjednodušené blokové schéma procesoru MPC5200

Tímto bych ukončil stručný přehled vlastností tohoto procesoru, pro zájemce o více informací odkazuji na firemní dokumentaci firmy Freescale ([1]).

2.1.2 Embedded modul Shark

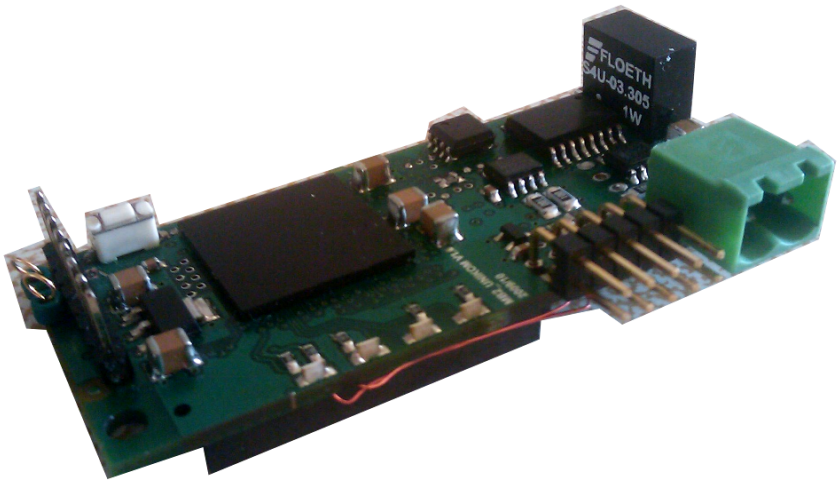


Embedded modul Shark ([2]) je založen na popsaném procesoru MPC5200B. Modul je postaven na základě podobného modulu německé firmy TQ Components ([3]) s názvem TQM5200 a je tak s ním pinově kompatibilní, další vlastnosti už jsou však upraveny pro plánované použití.

Vlastnosti modulu Shark

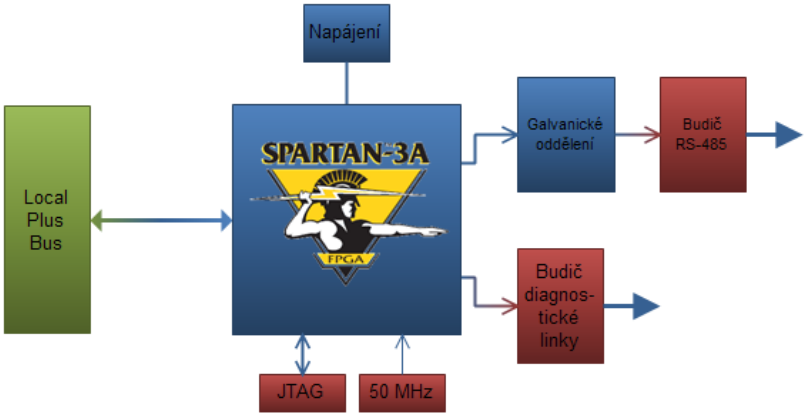
- Procesor: *MPC5200B*
- RAM: *až 128 MB SDRAM, 132 MHz*
- Flash: *až 64 MB NOR Flash*, k dispozici i 128 kB FRAM (náhrada EEPROM)
- Osazený ethernetový budič (PHY) - stačí přidat jen oddělovací transformátor a konektor
- Všechny I/O piny vyvedeny na dvojici 120 pinových board-to-board konektorů
- Potřeba pouze jednoho napájecího napětí *3.3 V*
- Rozměry: *80 x 60 x 8 mm*

2.2 FPGA Modul UNIKOM



Modul UNIKOM byl navržen pro realizaci rychlé sériové sběrnice PIRANHA. Základ modulu tvoří FPGA z rodiny Spartan-3A firmy Xilinx Inc. o velikosti 1,4 mil. ekvivalentních hradel, zdroj hodinového signálu 50 MHz, zdroj napětí pro FPGA a budič sběrnice RS-485, který je galvanicky oddělen od ostatních částí systému. Na následujícím obrázků je ukázáno blokové schéma modulu. Za zmínku stojí také fakt, že modul je pinově kompatibilní s moduly firmy Hilscher pro realizaci průmyslových rozhraní typu ProfiNet, ProfiBus, PowerLink a dalších. Existuje zde však odlišnost v podobě využití pinů konektoru, které moduly Hilscher nevyužívají a v použití diagnostické sériové linky pro účely konfigurace FPGA procesorem PowerPC.

FPGA v sobě integruje věškerou logiku řízení sběrnice PIRANHA a logiku pro připojení na sběrnici LocalPlus Bus (zkráceně LPB) procesoru PowerPC. Sběrnicí LPB se zabývá následující kapitola této práce. Procesoru PowerPC se FPGA jeví jako blok ve společné paměti na definované adrese, přes tuto část paměti probíha věškerá výměna dat mezi PowerPC a FPGA.



Obrázek 2.2: Blokové schéma modulu UNIKOM

2.3 Sběrnice PIRANHA

Sběrnice PIRANHA je rychlá sériová sběrnice použitá pro komunikace s lokálními nebo vzdálenými moduly vstupů / výstupů v průmyslových řídících systémech. Předpokládanému využití se podřizují požadavky na tuto sběrnici

- Vysoká rychlost sběrnice 20 MBps - nutné pro požadavek rychlé odezvu na čtení stavu vstupů
- Robustnost sběrnice - možnost komunikace při plné rychlosti na vzdálenost až 300 m v průmyslovém prostředí
- Sběrnice musí být použitelná jak pro malé moduly vstupů / výstupů umístěných na společné liště s procesorovým modulem tak i pro samostatné moduly určené k montáži na různá místa technologického celku
- V případě lokálních modulů musí sběrnice umožňovat automatickou konfiguraci a detekci připojených modulů na sběrnici - diagnostická linka

Na základě těchto požadavků byla navržena sériová sběrnice jejímž základem je fyzická vrstva postavená na bázi standardu RS-485. Sběrnice je koncipována jako Master-Slave, na principu dotaz-odpověď, je tedy použita varianta dvouvodičové RS-485 a tudíž polo-duplexního přenosu.

Přenosy na sběrnici jsou vždy zasazeny do přenosového rámce, který lze vyjádřit

následující strukturou

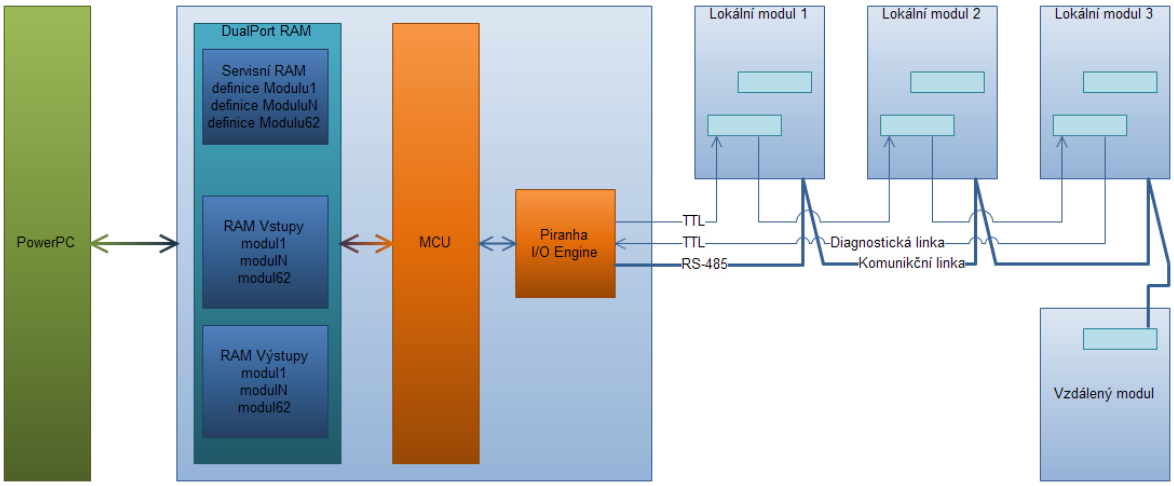
STX	(Řídící slova nebo data)	**...**	**ETX**	(**CRC**)

Význam znaků je následující

- **STX** - startovní slovo, obsahuje adresu modulu se kterým chceme komunikovat
- Po startovním slově následují volitelně řídící slova nebo data
- **ETX** - konec rámce, příslušným bitem je indikováno zda-li se ještě vysílá kontrolní CRC součet
- **CRC** - kontrolní CRC slov od STX až po ETX včetně

Všechna slova na komunikační a diagnostické sběrnici jsou devítibitová, kodóvání pro komunikační sběrnici je NRZI a pro diagnostickou sběrnici Manchester.

Na následujícím obrázků je znázorněno typické propojení sběrnice Piranha s třemi lokálními moduly a jedním vzdáleným včetně bloků uvnitř FPGA.



Obrázek 2.3: Schéma systému se sběrnicí PIRANHA

Na obrázku je patrná topologie jednotlivých sběrnic PIRANHA, komunikační linka je sběrnicové topologie, čili linka vždy vede od jednoho zařízení k druhému bez vzniku větvení. Diagnostická linka je založena na dlouhém posuvném registru vedoucím přes všechny lokální moduly, neaktivní modul provádí bypass tohoto registru. Délka posuvného registru v modulu je osm bitů, na tomto základě je možné automaticky určit

počet modulů na sběrnici. Tato detekce je potom podobná jako v případě sběrnice IEEE 1149.1, známé jako JTAG.

Sběrnice je navržena s šestibitovým adresováním zařízení, to umožňuje celkem 64 adres. Do systému je možno připojit až 62 zařízení, 2 adresy jsou definovány jako speciální. Zařízení s adresou 0 znamená nenaadresované, příkazy odeslané na adresu 0 jsou zařízením zpracovány, není ale odesílána žádná odpověď. Toto je využito při startu systému, kdy se pomocí diagnostické linky provede přiřazení adres jednotlivým zařízením. Pokud zařízení má již přiřazenu adresu z intervalu 1 až 62 na adresu 0 nereaguje. Adresa 63 je vyhrazena jako všesměrová, zpracovávají ji všechna zařízení na sběrnici včetně nenaadresovaných.

Přiřazení adres provadí master pomocí obou sběrnic, komunikační i diagnostické. Postup adresace je popsán následujícími kroky

1. Nejdříve je na diagnostickou linku vyslána sekvence 496 logických nul, toto zajistí vynulovaní posuvných registrů všech modulů na lince - tento krok není nutné provádět po připojení napájení, registry se automaticky nulují
2. Na diagnostickou linku vyšleme danou sekvenci osmi bitů pro aktivaci prvního modulu
3. Po komunikační lince vyšleme rámec pro nastavení adresy modulu s cílovou adresou 0, protože je první modul naadresován diagnostickou linku, může i odpovídat, tímto postupem jsme prvnímu modulu přiřadili adresu 1
4. Na diagnostickou linku vyšleme sekvenci osmi nul, tím deaktivujeme první modul a zaktivujeme druhý v pořadí
5. Opět po komunikační lince vyšleme rámec pro nastavení adresy jako v kroku 3, máme naadresovaný druhý modul
6. Kroky 4 a 5 opakujeme tak dlouho, dokud master nepřečte z diagnostické linky aktivační sekvenci, tím máme pro všechny moduly nastavenou komunikační adresu
7. Nyní můžeme přes známé adresy získat typy osazených modulů a pokračovat v jejich inicializaci

Kapitola 3

LocalPlus Bus, FPGA

3.1 LocalPlus BUS

3.1.1 Obecný popis

LocalPlus Bus je externí paměťová sběrnice procesoru MPC5200. Sběrnice umožňuje připojení bootovacích pamětí, pamětí typu ROM, RAM a paměťově mapovaných periferií. Řadič sběrnice poskytuje osm nezávislých paměťových oblastí a pro každou z nich je k dispozici samostaný signál *Chip select*. Řadič umožňuje nezávislé nastavení módu sběrnice pro danou oblast, lze vybírat z těchto možností

- Šířka datové sběrnice
  - 8, 16 nebo 32 bitů
- Šířka adresové sběrnice
  - 8 až 25(27) bitů
- Multiplexovaný nebo přímý režim sběrnice
- Programovatelný počet Wait stavů, Dead cyklů a záměnu pořadí bytů

Řadič poskytuje sdílený paměťový region použitý pro start systému (BootCS) společný s regionem 0 (CS0). Po resetu je v závislosti na stavu konfiguračních pinů namapován na adresu *0x00000000* nebo *0xFFF00000*, nastaven režim sběrnice (multiplexovaný nebo přímý) a pořadí bytů. V tomto případě je omezený výběr šířky sběrnic, lze použít tyto kombinace

- Přímý režim

  - 8-bitová datová sběrnice, 24-bitová adresová sběrnice
  - 16-bitová datová sběrnice, 16-bitová adresová sběrnice
- Multiplexovaný režim - adresa je vždy 25-bitová, lze volit jen šířku datové sběrnice
  - 16-bitová
  - 32-bitová

Při návrhu systému je nutné brát v potaz tato omezení a použít paměti a režim sběrnice takový, aby odpovídal nebo byl kompatibilní s některým z výše uvedených. Po startu je pak možné již korektně inicializovat region 0 podle skutečných požadavků připojení paměti a přepnutí ze startovacího regionu na nultý.

Řadič je připojen na programovatelný řadič DMA BestComm a umožňuje tedy datové přenosy z externích pamětí bez účasti procesoru. Paměťový subsystém bohužel nepodporuje nezarovnané přístupy, to je ale v této kategorii procesorů standartem. Řadič, který by uměl nezarovnané přístupy by byl výrazně komplikovanější na návrh jak vlastního hardware tak překladače. Dále by způsoboval komplikované připojení obvodů s datovou sběrnicí širší než osm bitů.

3.1.2 Signály

Abychom mohli dále vysvětlit fungování sběrnice LPB je nutné si nejdříve ujasnit, jaké signály nám poskytuje. Jako každá externí sběrnice obsahuje tři základní typy signálů - datovou, adresovou a řídící sběrnici. Podrobněji jsou rozepsány v tabulce 3.1 (značení signálu je dle katalogového listu k MPC5200)

Tabulka 3.1: LPB - používané signály

Signál	Směr přenosu pohled PowerPC	Význam
CS[7:0]	O	Výběr čipu, každá paměťová oblast má jeden signál CS, aktivní v log.0
R/W	O	Indikuje požadovaný směr operace, čtení = 1 / zápis = 0
EXT_AD[31:0]	I/O	obousměrná adresová a datová sběrnice, MSB = 31
ACK	I/O	Potvrzení transakce na sběrnici externím čipem (vstup) nebo indikace BURST cyklu (výstup)
TS	O	Začátek přenosu
OE	O	Povolení výstupů, řízení třístavového budiče datové sběrnice
TSIZ[2:0]	O	Šířka transakce, určuje velikost přenášených dat (1, 2 nebo 4 byty)
ALE	O	Aktivace záchytného registru pro zachycení adresy v případě multiplexované sběrnice, aktivace je na vzestupnou hranu

Funkce sběrnice EXT_AD se mění dle aktuálního módu činnosti sběrnice, toto je vysvětleno v následujících dvou sekcích. Funkce signálu OE není přímo generováná řadičem LPB ale pomocnou logikou, je určen následovně OE = CSx + (not R/W).
Sběrnice LPB odpovídá značení PCI, čili 31 je nejvíce významný bit zatímco 0 je nejméně významný. Čtecí nebo zapisovací cykly, ale odpovídají pořadí bytů *Big endian*, čili na nižší adrese je významnější byte.

3.1.3 Přímý mód

Přímý mód nebo také nemultiplexovaný. V tomto režimu dochází k rozdělení signálů EXT_AD na samostanou datovou a adresovou sběrnici, z toho je jasně patrné, že kombinace šířky datové a adresové sběrnice musí být menší než 32 signálů, pokud tomu

tak není dochází k použití pinů určených pro jinou funkci. Řadič podporuje kombinace uvedené v tabulce 3.2

Tabulka 3.2: Přímý režim LPB - možné režimy

Adresa	Data	Počet pinů	Velikost	Režim	Popis
8	8	16	256 B	Small	
8	16	24	256 B	Small	
16	8	24	64 kB	Small	
16	16	32	64 kB	Small	
24	8	32	16 MB	Medium	
24	32	56	16 MB	Most graphics	*1
26	8	34	64 MB	Large	*1
26	16	42	64 MB	Large	*1

*1 - adresová sběrnice nebo její část je v této kombinaci vyvedena na signály PCI, není ji tedy možno použít

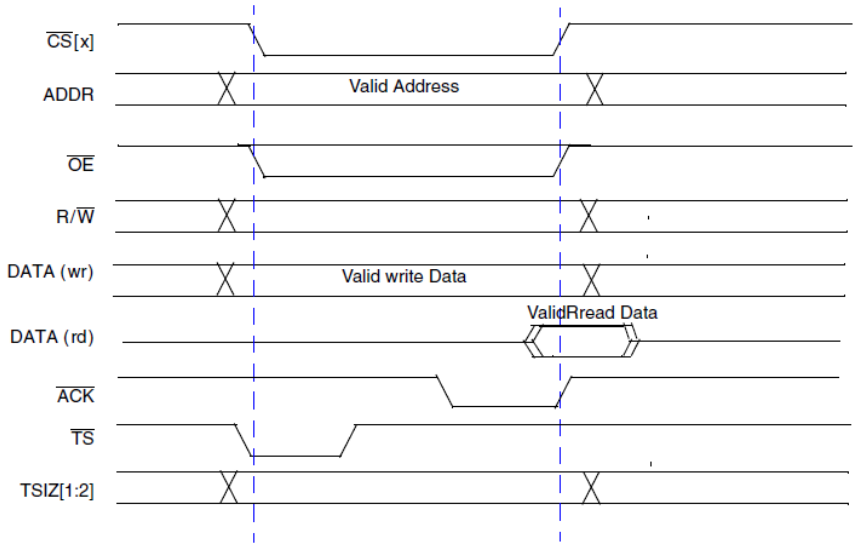
V tomto režimu sběrnice mají signály TSIZ pouze 2 bity a definují zda se přenáší 1, 2 nebo 4 byty. Jejich význam je uveden v tabulce 3.3. V případě čtení širšího slova než je šířka datové sběrnice, dochází k vícenásobnému přístupu na sběrnici. Na příkladu, máme 16-bitový port a čteme 32 bitů, dojde tedy ke dvěma čtením po 16 bitech. Naopak v případě čtení menšího počtu bitů než je šířka sběrnice, dochází k využití jen části sběrnice dané pořadím bitů. Na příkladu, máme 32-bitový port a čteme třetí byte, dojde tedy k vystavení adresy s nulovými spodními dvěma bity a s datové sběrnice se berou v potaz jen data z třetího bytu (čili bity 16 až 23).

Tabulka 3.3: TSIZ přímý mód

TSIZ1	TSIZ2	Šířka transakce
0	1	1 B
1	0	2 B
0	0	4 B

3.1.3.1 Přístup na sběrnici

Přenos dat začíná vystavením adresy na adresovou sběrnici a nastavením signálů R/W a TSIZ. V případě zápisového cyklu se vystaví platné data na datovou sběrnici. Poté se provede krátký negativní puls na signálu TS indikující začátek cyklu a následně aktivace signálu CS. Nyní paměťové zařízení provádí danou operaci. Po uplynutí nastaveného počtu Wait stavů nebo vzestupné hraně na signálu ACK dochází k deaktivaci signálu CS, v případě čtecího cyklu jsou v tuto chvíli přečtena správná data z datové sběrnice. Tím celý cyklus končí.



Obrázek 3.1: Časový diagram přístupu na LPB v přímém režimu

3.1.4 Multiplexovaný mód

V tomto režimu dochází k multiplexování adresové a datové sběrnice na společných signálech EXT_AD. Tento režim díky většími počtu signálu umožňuje použití větších šířek datové sběrnice společně s velkými kapacitami pamětí. Toho je dosaženo za cenu snížení propustnosti sběrnice jelikož na jednu operaci je třeba dvou cyklů sběrnice. Řadič v tomto režimu podporuje libovolné kombinace šířek adresové a datové sběrnice. Adresová sběrnice může být široká 8, 16, 24 nebo 25 bitů, k těmto jsou ještě k dispozici dva bity pro výběr banky. Datová sběrnice může mít šířku 8, 16 nebo 32 bitů. Rozložení sběrnic na signálech EXT_AD zachovává pravidlo, že datová sběrnice se

rozšiřuje směrem od bitu 31 k bitu 0, 8-bitová sběrnice tedy obsadí pouze bity 31 až 24. Adresová sběrnice se naopak rozšiřuje normálním způsobem od bitu 0 směrem k bitu 31, 8-bitová tedy obsadí bity 7 až 0.

Díky multiplexování adresové a datové sběrnice musí být vně na sběrnici připojena pomocná logika, která zajistí zachycení a udržení adresy během datového cyklu. Příklad připojení je uveden dále v textu.

Stejně jako v případě přímého módu i zde existují bity TSIZ určující velikost přenosu, v tomto případě jsou ale 3. Jejich význam je patrný z tabulky 3.4

Tabulka 3.4: TSIZ multiplexovaný mód

TSIZ0	TSIZ1	TSIZ2	Šířka transakce
0	0	1	1 B
0	1	0	2 B
1	0	0	4 B

V případě adresové fáze je k dispozici 32 signálu, ale jen 25 bitů adresy. Zbylých 7 signálů je rozloženo dle tabulky 3.5

Tabulka 3.5: Využití nejvyšších 8 bitů adresové sběrnice

Bit	Název	Význam
24	A24	Adresový bit 24
25	Bank select bit 0	Výběr banky
26	Bank select bit 1	
27	0	Trvale v log.0
28	TSIZ2	Výběr velikosti přenosu
29	TSIZ1	
30	TSIZ0	
31	0	Trvale v log.0

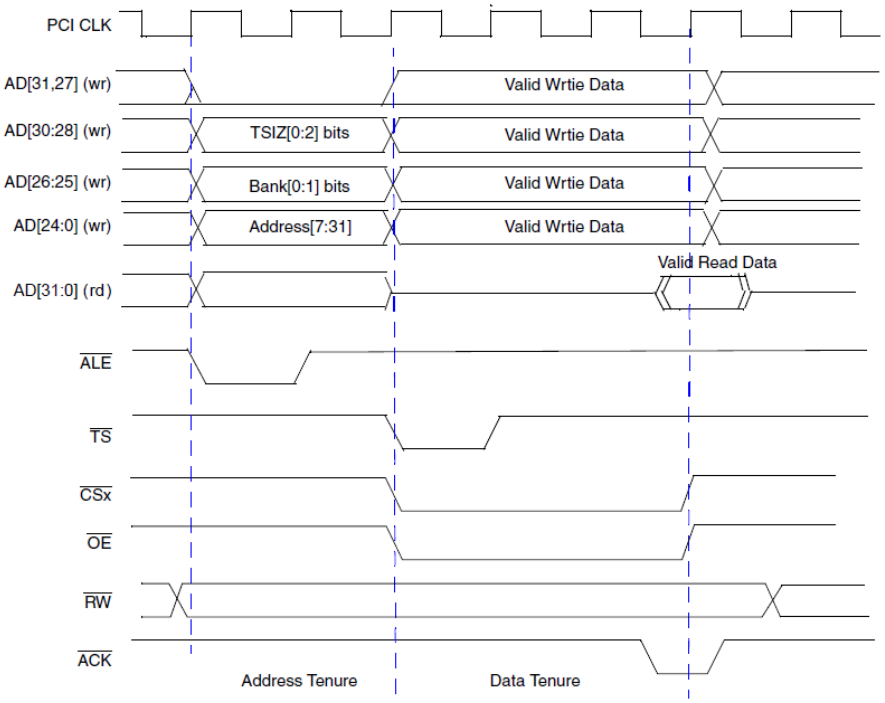
3.1.4.1 Přístup na sběrnici

Přenos na sběrnici je řízen hodinami sběrnicemi PCI a ke změnám výstupů dochází na vzestupnou hranu tohoto hodinového signálu. Časový diagram je na obrázku 3.2

Prvním krokem je nastavení signálu R/W pro určení směru cyklu sběrnice.

Dalším krokem je adresní fáze, kdy je na adresovou sběrnici vystavena maximálně 25-bitová adresa, 2 bity určující banku a bity TSIZ, zároveň je bit ALE nastaven do log. 0. V následném hodinovém cyklu je signál ALE vrácen zpět do log.1, tato vzestupná hrana indikuje platnost dat na sběrnici a je tedy signálem pro záchytný registr k uložení adresy a dalších bitů.

Druhým cyklem je datová fáze, jsou nastaveny signály TS a CS do log. 0, k tomu patřičně signál OE a taktéž vlastní data k zápisu na signály EXT_AD. Nyní paměťové zařízení provádí danou operaci. Po uplynutí nastaveného počtu Wait stavů nebo vzestupné hraně na signálu ACK dochází k deaktivaci signálu CS, v případě čtecího cyklu jsou v tuto chvíli přečtena správná data z datové sběrnice. Tím celý cyklus končí.

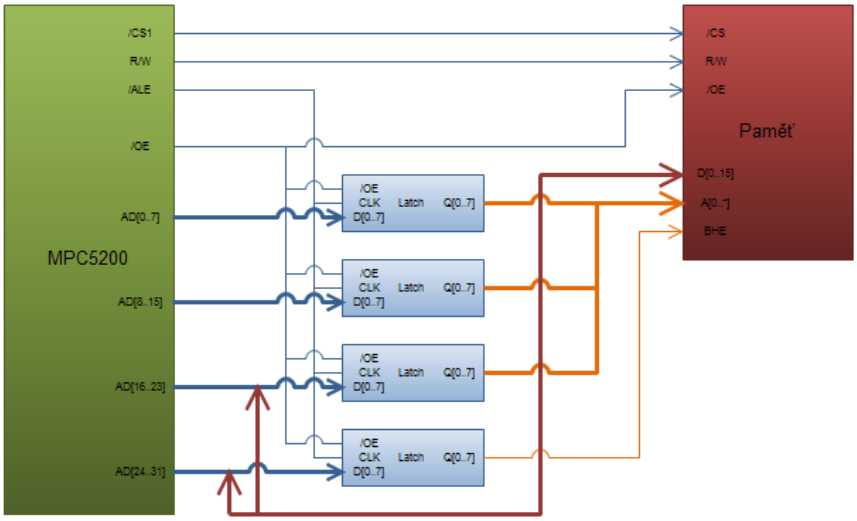


Obrázek 3.2: Časový diagram přístupu na LPB v multiplexovaném režimu

3.1.4.2 Příklad realizace připojení

Zde uvedu na příkladu připojení šestnácti-bitové paměti SRAM na multiplexovanou sběrnici LPB. Základem je použití záchytných registrů, latchů, pro zachycení stavu adresové sběrnice během adresní fáze, to je na obrázku znázorněno bloky s názvem Latch. Dále je nutné propojit signály CS, R/W a OE s odpovídajícími signály paměti.

Signál CS řídí aktivitu připojené paměti, pokud je v log.1 pamět nereaguje na aktivitu ostatních signálů, toto je výhodné během adresní fáze. Signálem OE řídíme třístavový budič sběrnice v paměti a tím určujeme kdo řídí datovou sběrnici, zda-li LPB nebo pamět. Posledním krokem je připojení datové sběrnice paměti přímo na signály EXT_AD sběrnice LPB.



Obrázek 3.3: Příklad připojení 16-bitové paměti na LPB v multiplexovaném režimu

3.1.5 Programový model LPB

Tabulka 3.6: sablona

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

V předcházejících kapitolách jsem ukázal činnost sběrnice LPB na hardwarové úrovni, v následující části popíši funkci řadiče z pohledu aplikace, která chce využívat periferii připojenou na LPB. K řízení LPB slouží několik registrů řadiče. Pomocí těchto registrů určujeme mód činnosti pro každou paměťovou oblast a její umístění v 32-bitovém adresním prostoru procesoru PowerPC.

Základní adresou všech registrů v procesoru MPC5200 je registr MBAR, ten definuje horních 16 bitů základní adresy, tento registr však trpí drobným problémem, jeho adresa je určena sebou samým. Pokud tuto základní adresu neznáme, nelze ji najít. Po resetu je registr MBAR inicializován na hodnotu 0x00008000, nachází se tedy na adrese 0x80000000, umístění v polovině adresního prostoru je nepraktické a tak U-Boot při startu přesouvá tento registr na adresu 0xF0000000. Dále v textu budu uvádět všechny adresy registrů relativně k adrese dané MBAR, adresa bude uváděna za názvem registru v kulatých závorkách. Číslování bitů odpovídá Big endianu, bit s číslem 0 je nejvíce vlevo.

Základním registrem pro LPB je *Chip select control register* (0x0318), tento registr obsahuje jediný bit 7 - ME, *Master enable*. Pro fungování celého řadiče musí být v 1, pokud je 0 nedochází ke generování signálů na sběrnici LPB. Výjímku tvoří paměťová oblast pro start systému - BOOTCS, tu nastavení bitu ME neovlivňuje. Po resetu má hodnotu 0.

Dalším registrem, který nás zajímá je *IPBI Control register* (0x0054), v něm najdeme bity povolující jednotlivé paměťové oblasti. Struktura registru je v tabulce 3.7

Tabulka 3.7: IPBI Control register (0x0054)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Vyhrazené				CS7 Ena	CS6 Ena	BootCS Ena	Vyhrazené			CS5 Ena	CS4 Ena	CS3 Ena	CS2 Ena	CS1 Ena	CS0 Ena
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Vyhrazené															WSE
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Bity CS* povolují příslušný paměťový region, BootCS je aktivní po resetu a je potřebný pro start systému. Bit WSE povoluje použití Wait stavů a pro rychlosti sběrnice vyšší než 66 MHz by měl mít vždy hodnotu 1.

Již jsem zmínil, že před použitím paměťové oblasti je nutné specifikovat, kde se daná oblast v paměti nachází, o to se stará vždy dvojice registrů pro každou oblast. Jde o registry CS*Start a CS*Stop. Registry mají následující offsety

Tabulka 3.8: Adresy registrů CS*Start a CS*Stop

Oblast	CS*Start	CS*Stop
0	0x0004	0x0008
1	0x000C	0x0010
2	0x0014	0x0018
3	0x001C	0x0020
4	0x0024	0x0028
5	0x002C	0x0030
Boot	0x004C	0x0050
6	0x0058	0x005C
7	0x0060	0x0064

Obsah registru definuje vyšších 16 bitů adresy, mezi kterými daná oblast leží. Platný rozsah adres je dán takto, nejnižší adresa je dána pomocí CS*Start doplněné zprava 0x0000 a poslední platná adresa je CS*Stop doplněná zprava jedničkami, příklad CS0Start = 0x0000FC00, CS0Stop = 0x0000FFFF, platný rozsah adres pro oblast 0 je tedy 0xFC000000 až 0xFFFFFFFF. Struktura registrů je následující

Tabulka 3.9: Registry CS*Start a CS*Stop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Vyhrazené															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Bázová adresa / Koncová adresa															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Pro konfiguraci paměťových oblastí slouží registry *Chip select * configuration register* (0x300 + n*4). Tento registr definuje mód činnosti sběrnice pro danou oblast, má nasledující strukturu

Tabulka 3.10: Chip select configuration register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
WaitP								WaitX							
0	0	0	0	0	0	0	0	cfg	cfg	cfg	cfg	cfg	cfg	cfg	cfg
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
MX	Vyhr.	AA	CE	AS		DS		Bank		WTyp		WS	RS	WO	RO
cfg	1	1	1	cfg	cfg	cfg	cfg	0	0	0	0	0	0	0	1

Význam bitů je následující

- *WaitP, WaitX* - Definice délky Wait stavů, význam je dán dvojicí bitů *WTyp*
- *MX* - Mód sběrnice buď multiplexovaný (=1) nebo přímý (=0)
- *AA* - Určuje, zda se používá (=1) vstup ACK, pomocí něho lze zkrátit počet wait stavů

- *CE* - Individuální povolení daného CS, pro povolení činnosti musí mít hodnotu 1
- *AS* - Určuje šířku adresové sběrnice, 00 → 8b, 01 → 16b, 10 → 24b, 11 → >25b
- *DS* - Určuje šířku datové sběrnice, 00 → 8b, 01 → 16b, 11 → 32b
- *Bank* - Hodnota bitů A26 a A25 adresové sběrnice v případě multiplexovaného módu LPB
- *WTyp* - Určuje význam nastavení WaitP a WaitX
  - 00 - WaitX použito pro čtení i zápis, WaitP se ignoruje
  - 01 - WaitX použito pro čtení, WaitP pro zápis
  - 10 - WaitX použito pro čtení, WaitP/WaitX jako 16-bitová hodnota pro zápis
  - 11 - WaitX/WaitP použito jako 16-bitová hodnota pro čtení i zápis
- *WS* - V případě hodnoty 1 provádí záměnu pořadí bytů při zápisu
- *RS* - Stejné jako WS, jen pro případ čtení
- *WO* - Příznak pro zařízení určené pouze pro zápis, pokus o čtení z takovéhoto zařízení skončí vyvoláním vyjímky chyby sběrnice
- *RO* - Stejné jako WO, jen pro případ čtení

Posledním, pro nás okrajovým, je registr pro nastavení Dead cyklů sběrnice. Určuje počet cyklů, po které je signál CS držen v aktivní úrovni po skončení cyklu čtení. Toto je potřeba pro zařízení, které potřebují jistý čas k převedení své datové sběrnice do třetího stavu. Registr se jmenuje *Chip Select Deadcycle Control Register* a je na adrese 0x032C. Struktura je následující

Tabulka 3.11: Dead cycle configuration register

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Vyhrazené		DC7		Vyhrazené		DC6		Vyhrazené		DC5		Vyhrazené		DC4	
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Vyhrazené		DC3		Vyhrazené		DC2		Vyhrazené		DC1		Vyhrazené		DC0	
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

Bity DC* přímo specifikují počet Dead cyklů v rozsahu od nuly do tří.

Pro využití požadované paměťové oblasti je nutné provést zápis příslušných hodnot do předchozích registrů. Pro příklad, paměťovou oblast 2 budeme chtít namapovat do rozsahu adres 0xFB000000 až 0xFB00FFFF, datová sběrnice bude 16-bitová, adresová také. Využijeme multiplexovaný mód s 8-mi čekacími stavy. V příkladu je využito syntaxe příkazů bootloaderu U-Boot.

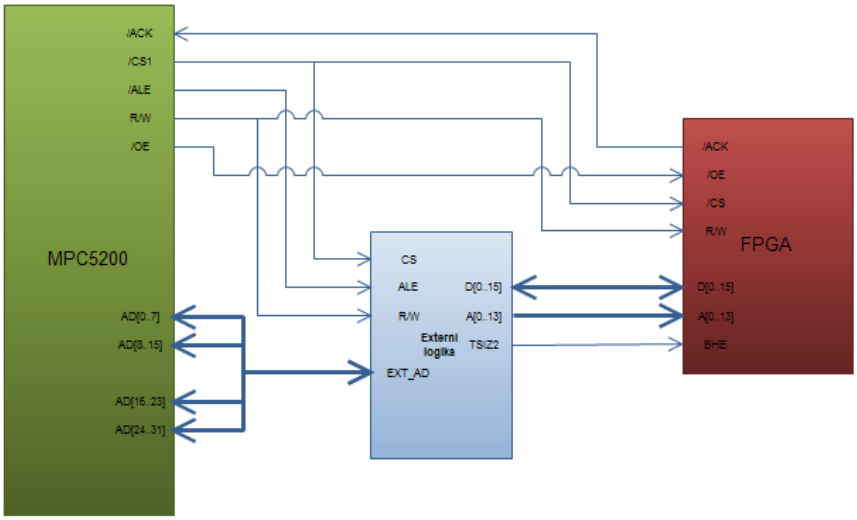
```
mwand f0000054 fffbffff
mw f0000014 0000fb00
mw f0000018 0000fb00
mwor f0000054 00040000
mw f0000308 0008d530
mw f0000318 01000000
```

Následuje vysvětlení příkazů, první provede deaktivaci oblasti 2, následující dva příkazy namapují oblast do paměti na požadovanou adresu, poté dojde opět k aktivaci oblasti 2, nastavení módu činnosti sběrnice a globální aktivaci paměťových oblastí. Po této inicializaci je paměťová oblast nakonfigurována a k dispozici.

3.1.6 Propojení PowerPC a FPGA

V této sekci popíši připojení FPGA na sběrnici LPB a realizaci vnitřní logiky FPGA tak, aby se chovala jako paměťový obvod. Způsobů propojení je mnoho, lze využít různá rozhraní procesoru PowerPC - SPI, I^2C nebo již popisovanou sběrnici LPB. S ohledem na rychlost a jednoduchost implementace je zvolena sběrnice LPB. Výběr módu sběrnice je pevně daný, na základnové desce pro modul Shark a Unikom je využita multiplexovaná sběrnice a současně jsou na ní osazeny záchytné registry. Modul Unikom má tedy již k dispozici samostatnou 16-bitovou datovou a 14-bitovou adresovou sběrnici a řídící signály CS, R/W, OE, ACK a TSIZ2. Signál ACK není implementací FPGA využit.

Vzhledem k předpokládanému využití modulu a existenci 16-bitové datové sběrnice je dekódování velikosti přenosu zjednodušeno pouze na bit TSIZ2, pokud se provádí 16-bitový přístup má hodnotu log.0, pokud 8-bitový pak má hodnotu log.1 a v kombinaci s nejnižším bitem adresy zjistíme o který byte se jedná.



Obrázek 3.4: Blokové schema realizovaného propojení PowerPC a FPGA

Na předchozím obrázku 3.4 je patrné, že se příliš neliší od příkladu připojení paměti. Blok *Externí logika* realizuje záchytné registry pro zachycení adresy a třístavové registry pro datovou sběrnici. Návrh logiky uvnitř FPGA musí odpovídat časování sběrnice LPB, zde dochází díky hardwarovému návrhu ke komplikaci tím, že LPB a FPGA jsou v různých časových doménách, LPB běží na 133 či 66 MHz, kdežto FPGA má k dispozici hodinový signál 50 MHz.

Základem pro komunikaci procesoru PowerPC a modulů na sběrnici PIRANHA je fungující dvou-bránová pamět mezi procesorem PowerPC a mikrokontrolérem uvnitř FPGA řídícím sběrnici PIRANHA. Jako první krok tedy bylo nutné realizovat v FPGA blokovou RAM připojenou na LPB, tím vznikne část ze strany PowerPC, která se poté vloží do výsledného projektu řadiče sběrnice PIRANHA. Pro připojení na LPB si musíme ujasnit, které cykly budou na sběrnici probíhat a jak bude takový cyklus vypadat. Máme 16-bitovou datovou sběrnici, musíme tedy vyřešit cyklus čtení/zápisu 16-bitové hodnoty a čtení/zápis 8-bitové hodnoty, 32-bitová hodnota jsou dva 16-bitové zápisy, které řeší řadič LPB za nás a tudíž je mi řešit nemusíme. Pro názornost je na následujících obrázcích uvedeno rozložení bitů na datové a adresové sběrnici pro řešené případy, čte/zapisuje se hodnota 0x1234 na sudou adresu.

Tabulka 3.12: Čtení/zápis 16-bitové hodnoty

A[13..1]	A0	TSIZ2	D[15..8]	D[7..0]
xxx	0	0	0x12	0x34

Tabulka 3.13: Čtení/zápis 8-bitové hodnoty

A[13..1]	A0	TSIZ2	D[15..8]	D[7..0]
xxx	0	1	0x12	
xxx	1	1		0x34

Z předchozích tabulek vidíme, že cyklus čtení z FPGA je jednoduchý, stačí vystavit na datovou sběrnici 16-bitovou hodnotu ze sudé adresy a řadič LPB si ji vezme buď celou, nebo pouze potřebný byte, např. v případě čtení bytu z liché adresy jsou data na signálech D[15..8] ignorovány a v případě bytu ze sudé adresy je tomu právě naopak. Situace se zápisovým cyklem je již složitější, výběr konkrétního bytu musíme řešit sami. Povolení zápisu hodnoty musíme tedy řešit logickou funkcí následujících signálů - CS, R/W, TSIZ2 a A0.

Pro určení logické funkce si sestavíme redukovanou tabulku stavů. Signály pro řízení zápisu vyššího a nižšího bytu si označíme jako *we_h* a *we_l*, které jsou aktivní v log.1.

Tabulka 3.14: Tabulka stavů pro signály we_h a we_l

$\overline{CS}$	R/$\overline{W}$	TSIZ2	A0	we_h	we_l
1	x	x	x	0	0
0	1	x	x	0	0
0	0	0	x(0)	1	1
0	0	1	0	1	0
0	0	1	1	0	1

x(0) znamená, že na hodnotě signálu sice nezáleží, ale z principu řadiče LPB je v tomto případě vždy v nule.

Nad touto tabulkou není ani třeba provádět složité minimalizace, vidíme následující. Aby mohlo k zápisu dojít, musí být CS v log.0, výsledné funkce musí být v lo-

gickém součinu s negací CS, zbytek funkce je pak již snadno patrný. Výsledné logické funkce tedy jsou $we\_h = \overline{R/W + A0} \,.\, \overline{CS}$ a $we\_l = \overline{R/W + (TSIZ2 \oplus A0)} \,.\, \overline{CS}$

Dalším krokem, který musíme vyřešit je rozdílné časování LPB a FPGA. Hlavní problém způsobuje zapojení záchytných registrů na základnové desce, jejich hodinový vstup je připojen na signál ALE a povolení výstupů na signál CS. Toto ale znamená, že ve chvíli kdy se aktivuje signál CS není k dispozici ustálená adresa na adresové sběrnici, v prvotní verzi se toto projevovalo přepisováním hodnoty na nulté adrese v paměti. Řešení tohoto problému spočívá ve zpoždění CS signálu, v konkrétním případě o 3 hodinové takty FPGA. Toto zpoždění je realizováno 3-bitovým posuvným registrem.

3.1.6.1 Návrh logiky uvnitř FPGA

Návrh FPGA byl proveden v jazyce VHDL a syntesizován ve vývojovém prostředí ISE WebPack verze 11 firmy Xilinx Inc. Návrh každé ucelené jednotky v jazyce FPGA se skládá ze dvou částí, v případě komponenty ze tří. První částí je návrh takzvané entity, což je definice jak se jednotka tváří navenek. Entita se skládá z definice signálů. Lze si ji představit jako popis pinů integrovaného obvodu. Druhou částí je architektura, ta vždy patří k již definované entitě. Pro jednu entitu může existovat více architektur, obráceně to samozřejmě nejde. Architektura již popisuje vztahy a vazby mezi signály definovanými entitou a tím určuje její chování. V případě, že vytváříme komponentu, čili blok pro opětovné použití, je třeba ještě napsat její definici, která ve většině případů kopíruje definici entity.

Návrh v jazyce VHDL je ve své podstatě hiearchický. Nejdříve vytvoříme menší části a ty postupně sdružujeme do větších celků, až se dostaneme k entitě, která je na vrchu a je jediná. Této entitě poté říkáme *TopEntity*, ke které se v prostředí ISE WebPack váže neméně důležitý soubor s příponou *ucf*, což znamená *User Constraint File*. Tento soubor obsahuje definici elektrických a návrhových vlastností k daným signálům. V našem případě ho využijeme k přiřazení konkrétních pinů signálům entity a pro definici jejich napěťových standartů.

TopEntita v tomto případě vypadá následovně

```
1 entity TopEntity is
2   Port (
3     clk_i     : in  STD_LOGIC;
4     rst_i     : in  STD_LOGIC; -- active low
5     address_i : in  std_logic_vector(13 downto 0);
```

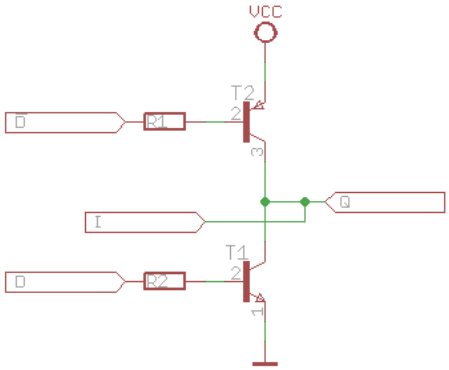
```
6       data_io    : inout std_logic_vector(15 downto 0);
7       lp_rw_i    : in std_logic;  -- r=1, w=0
8       lp_oe_i    : in std_logic;  -- active low
9       int_o      : out std_logic; -- active low
10      lp_ack_i   : in std_logic;
11      bhe_i      : in std_logic;  -- active low
12      cs_i       : in std_logic;
13
14      led1_o     : out  STD_LOGIC;
15      led2_o     : out  STD_LOGIC;
16      led3_o     : out  STD_LOGIC
17    );
18  end TopEntity;
```

Nyní, když máme definovány vstupy a výstupy můžeme začít vytvářet architekturu. Návrh jsem rozdělil do čtyř části

- Třístavový budič datové sběrnice
- Posuvný registr pro zpoždění signálu CS
- Kombinační logika pro generování signálů pro zápis
- Vlastní blokové paměti

3.1.6.2 Třístavový budič

Třístavový budič se nejčastěji používa ke spojení oddělených jednosměrných sběrnic, tedy samostatný vstup a výstup dat, na jednu obousměrnou sběrnici. K řízení směru se využíva signálu OE, který určuje jaká strana aktivně řídí sběrnici. Obvykle platí negativní logika kdy v log.0 sběrnici řídí periferie a v log.1 procesor či jiné zařízení v roli nadřízeného. Princip realizace třístavového budiče je na následujícím obrázku



Obrázek 3.5: Schéma třístavového budiče

V případě, že chceme mít na výstupu Q log.1 přivedeme na vstup D log.0 a na vstup /D log.0, pro opačnou hodnotu výstupu se stavy na D a /D otočí. Pokud na vstup D přivedeme log.0 a na /D log.1 jsou oba tranzistory zavřené a výstup Q může být zvenku změněn, jeho hodnotu pak máme na výstupu I.
Ve VHDL se takovýto budič vytvoří snadno. Signály jsou definovány takto, *data_io* je obousměrná sběrnice, *data_i* je z hlediska FPGA vstup dat a *data_o* je výstup dat.

```
1  -- simple tri-state buffer
2  data_io <= data_o when (lp_oe_i = '0') else (others => 'Z');
3  data_i <= data_io;
```

3.1.6.3 Posuvný registr

Posuvný registr se v jazyce VHDL definuje opět velice snadno, realizace je následující

```
1  -- definice cs2
2  signal cs2 : std_logic_vector(2 downto 0);
3
4  -- delay PowerPC chip select by 3 clk_i clocks
5  process(clk_i)
6  begin
7    if clk_i'event and clk_i = '1' then
8      cs2 <= cs_i & cs2(2 downto 1);
9    end if;
10 end process;
```

Funkci popíšeme následovně, posuvný registr je založen na 3 klopných obvodech (KO), kdy data posouváme zleva doprava. Každou vzestupnou hranou hodinového signálu clk_i se vezme výstup prvních dvou a zleva se doplní signálem ze vstupu cs_i. Takto získané tři bity se poté vloží do KO. Nejlépe bude ukazát na příkladu, pro začátek máme v KO samé nuly

- Stav KO: 000, na vstupu cs_i 1 → Nový stav KO: 100
- Stav KO: 100, na vstupu cs_i 0 → Nový stav KO: 010
- Stav KO: 010, na vstupu cs_i 0 → Nový stav KO: 001
- Stav KO: 001, na vstupu cs_i 0 → Nový stav KO: 000

Jako signál CS poté v FPGA používáme stav posledního KO, čili signál cs2(0), KO jsou počítány od 2 sestupně zleva doprava.

3.1.6.4 Kombinační logika

Postup jakým získat logické funkce pro řízení zápisů jednotlivých bytů jsme si odvodili v odstavci výše, zbýva nám je pouze přepsat do jazyka VHDL

```
1  -- definice signálů
2  signal mem_we_lo : std_logic;
3  signal mem_we_hi : std_logic;
4
5  -- přiřazení správných hodnot
6  mem_we_hi <= (not ( address_i(0) or lp_rw_i)) and (not cs2(0));
7  mem_we_lo <= (not ( (bhe_i xor address_i(0)) or lp_rw_i))  and (not cs2(0));
```

3.1.6.5 Instance blokových pamětí

Posledním krokem realizace v FPGA je připojení vlastních pamětí, jsou využity dvě paměti s osmibitovou datovou sběrnicí sloužící jako vyšší a nižší byte.

```
1  -- definice signálů
2  signal addrmem : std_logic_vector(10 downto 0);
3
4  -- přiřazení správných hodnot
5  addrmem <= address_i(11 downto 1);
6  ramb16_s18_inst_lo : ramb16_s9
7    port map (
8      do => data_o(7 downto 0),
9      di => data_i(7 downto 0),
10     dip => (others => '0'),
11     addr => addrmem,
12     clk => clk_i,
13     en => '1',
14     ssr => '0',
15     we => mem_we_lo
16   );
17 ramb16_s18_inst_hi : ramb16_s9
18   port map (
19     do => data_o(15 downto 8),
20     di => data_i(15 downto 8),
21     dip => (others => '0'),
22     addr => addrmem,
23     clk => clk_i,
24     en => '1',
25     ssr => '0',
26     we => mem_we_hi
27   );
```

Připojení je provedeno dle příkladu v nápovědě k ISE WebPack. Za zmínku stoji připojení adresové sběrnice, kdy na paměti je připojena adresová sběrnice posunutá

o 1 bit, bit A0 je zdánlivě ignorován. To vychází z dřívějšího odstavce, kdy jsme definovali chování sběrnice pro různé šířky přenosů. Pro operaci čtení signál A0 řešit nemusíme, o to se postará řadič LPB a pro případ zápisu je obsažen ve funkcích pro povolení zápisu.

3.1.7 Otestování na PowerPC

Otestování provedeme pomocí bootloaderu U-Boot, který v sobě zahrnuje nástroje pro manipulaci s pamětí. Inicializace LPB registrů je shodná s příkladem v sekci 3.1.5. Podle něho provedeme inicializaci a použitím příkazů *md* a *mw* otestujeme funkčnost například následujícím způsobem

```
md fb000000
mw fb000004 12345678
md fb000000
```

První příkaz vypíše obsah 256 bytů paměti od adresy 0xFB000000, druhý zapíše na adresu 0xFB000004 hodnotu 0x12345678 a nakonec pomocí třetího příkazu opět vypíšeme obsah paměti, kde uvidíme zapisovanou hodnotu.

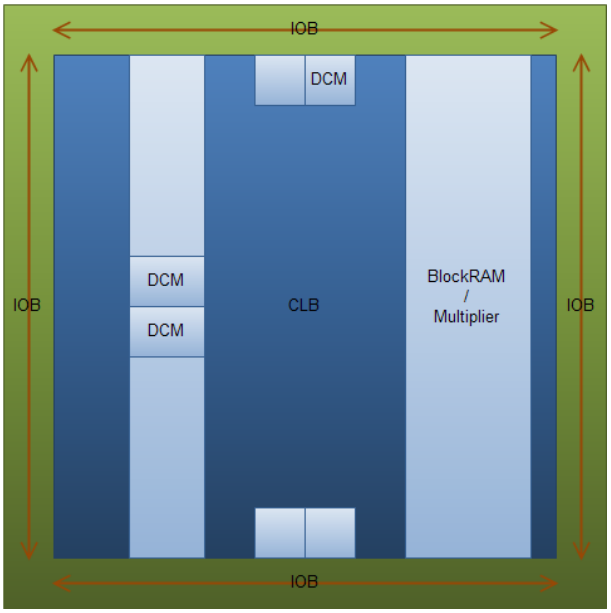
3.2 FPGA konfigurace

3.2.1 FPGA

FPGA je uživatelsky programovatelná součástka, její funkce není výrobcem pevně dána. Prvními programovatelnými obvody byly paměti ROM, adresová sběrnice tvořila vstupní proměnné a datová sběrnice výstupy. Obsahem paměti byla tabulka s definicí výstupní proměné, každý bit datové sběrnice odpovídal jedné funkci. Dalším krokem jsou obvody PAL/GAL, skládají se z programovatelné matice AND a pevné OR, vstupem jsou vstupy obvodu a na jejich výstupu jsou výstupní bloky obsahující navíc klopný obvod. Výstup tohoto bloku je vyveden na piny obvody a zároveň zpět do AND matice. Obvody PAL/GAL byly poměrně malé, vznikly tedy obvody CPLD. Jsou tvořeny několika několika obvody PAL/GAL a propojovací maticí mezi nimi. Pokud dáme dohromady velké množství CPLD, doplněné propojovací maticí a vstupně-výstupními bloky

vznikne FPGA. Současné řady FPGA obsahují i pokročilé funkční bloky jako například blokové paměti RAM, hardwarové násobičky, generátory hodin a další.

Typická struktura obvodu FPGA je na následujících dvou obrázcích. Na prvním je celkový náhled na rozložení bloků na ploše FPGA, na tom druhém je zvětšená část s vyznačením konkrétních bloků.



Obrázek 3.6: Struktura obvodu FPGA

Obrázek 3.7: Podrobnější pohled na strukturu obvodu FPGA

Z důvodu vysoké rychlosti obvodů FPGA, jsou všechny programovatelné propoje realizovány na bázi paměti RAM. Po připojení napájení je tedy nutné nejdříve provést jeho konfiguraci. To se provede nahráním bloku dat, který se nazývá BitStream. K tomuto účelu existuje u FPGA speciální rozhraní pro připojení tzv. platform flash, ze které si FPGA umí po startu načíst konfiguraci, po jejím skončení už tuto paměť nepotřebuje. Tato pamět je obvykle sériová, takže stači minimum vodičů a pro naprogramování nabízí rozhraní JTAG.

Výrobci FPGA obvykle umožňují i jiné způsoby konfigurace. Druhým základním je opět rozhraní JTAG, to má smysl ale pouze pro vývoj. Další metody rozlišují zda je FPGA tím, kdo řídí konfiguraci, čili je master nebo je ve funkci podřízeného, kdy konfiguraci řídí externí procesor. Metody taktéž můžeme rozdělit na paralelní a sériovou. Cílem zadání je navrhnout a realizovat metodu konfigurace FPGA procesorem PowerPC. Z důvodů minimalizace počtu propojů mezi FPGA a PowerPC jsem zvolil sériovou metodu a roli FPGA jako slave. Celý algoritmus konfigurace bude dopsán do bootloaderu U-Boot, kde již existuje jistý framework pro konfiguraci FPGA.

3.2.2 Konfigurace Slave Serial

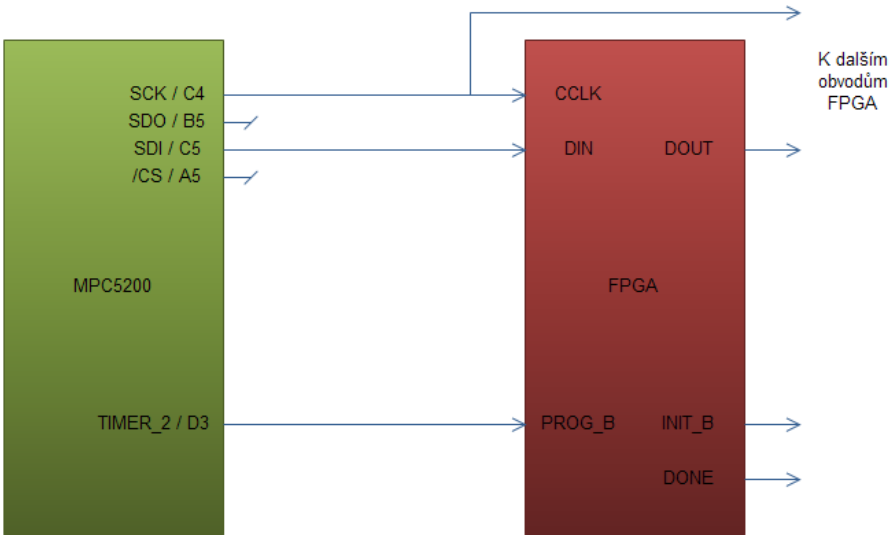
Jak již bylo zmíněno, při této metodě řídící procesor nahrává BitStream do FPGA sériově. Velikost BitStreamu je závislá na velikosti FPGA. Modul UNIKOM je založen na FPGA Spartan-3A firmy Xilinx Inc. proto zde pro názornost uvedu velikosti pro jednotlivé obvody této rodiny

Tabulka 3.15: Velikost BitStreamu pro rodinu Spartan-3A

Označení	Počet konfiguračních bitů
XC3S50A	437 312
XC3S200A	1 196 128
XC3S400A	1 886 560
XC3S700A	2 732 640
XC3S1400A	4 755 296

FPGA musí při startu zjistit, jaký režim konfigurace bude použit. K tomu účelu slouží trojice pinů označených M[2..0]. Pro volbu režimu slave serial musí mít všechny hodnotu log.1. Způsob propojení PowerPC a FPGA realizovaný v této práci je na

následujícím schématu.



Obrázek 3.8: Propojení FPGA s PowerPC pro účel konfigurace

Význam signálů je následující

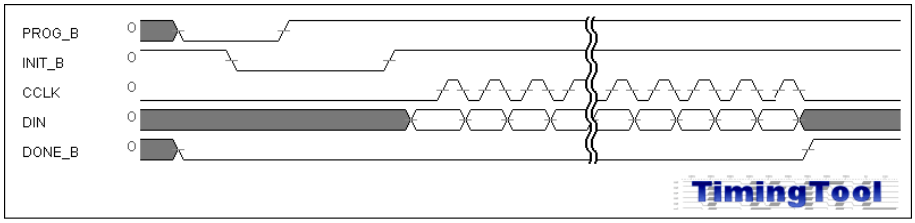
Tabulka 3.16: Význam signálu pro režim slave serial

Signál	Směr	Význam
CCLK	I	Vstup hodinového signálu
DIN	I	Vstup dat
DOUT	O	Výstup dat, využito v případě konfigurace více FPGA jedním masterem
PROG_B	I	Puls do log.0 o minimální délce 500 ns, způsobí restart konfiguračního procesoru
INIT_B	O	Indikace výmazu paměti nebo chyby při konfiguraci, signál je aktivní v log.0
DONE_B	O	Indikace úspěšnosti konfigurace, po celou dobu konfigurace jev log.0, po úspěšné konfiguraci je v log.1

Signály INIT_B a DONE_B nejsou v implementaci kvůli nedostatku pinů využity, signál INIT_B je nahrazen krátkým zpožděním po deaktivaci signálu PROG_B. Signál DONE_B se netestuje. Úspěšnost konfigurace lze otestovat i pomocí software v součinnosti s návrhem v FPGA, například přečtením dané adresy z FPGA nebo zápis se

zpětným čtením. Tento test takto stačí, nemůže nastat případ, kdy se FPGA nakonfiguruje pouze částečně.

Princip této metody konfigurace je jasněji patrný z následujícího časového diagramu



Obrázek 3.9: Časový diagram konfigurace FPGA

Proces komunikace začíná pulsem do log.0 na signálu PROG_B, během této doby přechází do log.0 i signál INIT_B. Po tomto musíme čtením čekat až signál INIT_B přejde zpět do log.1, tím je FPGA připraveno na příjem BitStreamu. Poté následuje vlastní odeslání BitStreamu, kdy hodnota bitu je platná na vzestupnou hranu hodinového signálu. Po odvysílání celého BitStreamu, musíme i nadále generovat hodinový signál až do doby, kdy přejde signál DONE_B do log.1. Tento poslední krok je také závislý na nastavení, se kterými byl BitStream vygenerován. Pro nás budou zajímavé dvě nastavení. Prvním je zdroj hodinového signálu pro rozběh FPGA, zde musí být zvolen hodinový signál, který je použit i pro konfiguraci, čili CCLK - to je onen důvod, proč musíme vygenerovat více hodinovým pulsů než je bitů v BitStreamu. Druhé nastavení se týka signálu DONE_B a určuje jeho chování po dokončení konfigurace. My využíváme, že přejde do log.1. Po přechodu signálu DONE_B do log.1 dochází k propojení signálu DIN a DOUT, tím můžeme nakonfigurovat více FPGA pomocí jednoho mastera.

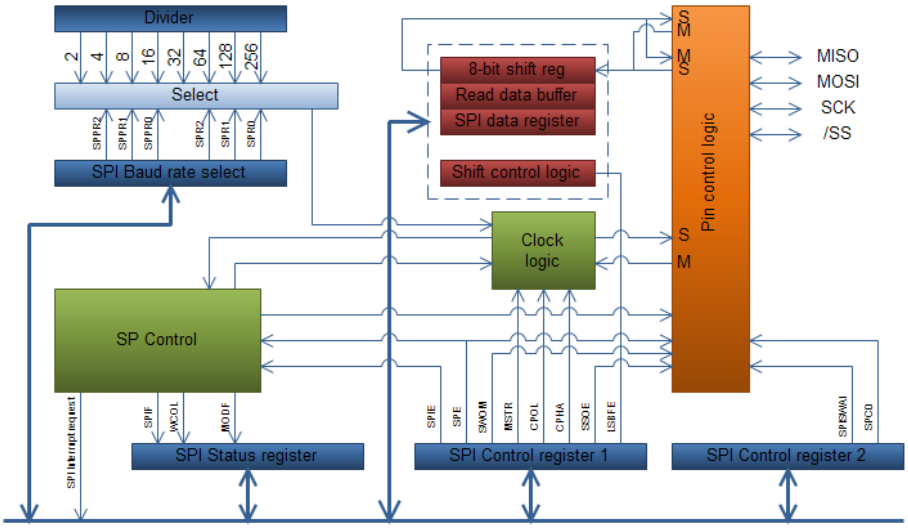
Na straně PowerPC bylo pro konfiguraci zvoleno rozhraní SPI, které je k dispozici jako periferie. Důvody pro jeho využití, U-Boot provádí odvysílání BitStreamu pomocí bitového řízení pinů, tento postup je však pomalý a zbytečně zatěžuje jádro procesoru. Využití SPI periferie nám umožňuje výrazně zkrátit dobu potřebnou na konfiguraci FPGA. Na odvysílání jednoho bytu potřebujeme pouze 1 zápis do registru oproti třicetidvěma při bitovém přístupu, také frekvence hodinového signálu může být větší. Doba trvání konfigurace je pak přibližně stejná, jako konfigurace z platform flash.

3.2.3 Popis SPI periferie

Periferie slouží pro sériovou plně duplexní, synchronní komunikaci mezi PowerPC a externím zařízením. Nejmenší přenášenou entitou je jeden byte, čili osm bitů. Stručně lze shrnout následující vlastnosti

- Umožňuje režim Master i Slave
- 4-vodičový režim (Vstup Výstup dat, hodinový signál, slave select) nebo 3-vodičový režim(obousměrná data, hodinový signál, slave select)
- Detekce chybových stavů
- Dvojitý buffer na přijatá data
- Nastavitelná fáze a polarita hodinového signálu
- Možnost použití signálů jako prosté GPIO

Na následujícím obrázku je znázorněné blokové schéma periferie.



Obrázek 3.10: Blokové schéma SPI periferie

Sběrnice SPI je jednoduchá sériová sběrnice hojně používaná pro komunikaci s různými typy obvodů jako jsou např. AD a DA převodníky, LCD displaye, digitální potenciometry a další. Její jednoduchost spočívá ve třech (čtyřech) signálech, jsou to následující

- MISO - *master in slave out*, vstup sériových dat na straně mastera
- MOSI - *master out slave in*, výstup sériových dat na straně mastera

- SCK - hodinový signál
- /SS - aktivace cílového obvodu - tento signál však není pro SPI nepodstradatelný, může fungovat i bez něho

Data jsou čtena a zapisována synchronně s hodinovým signálem, při jaké hraně a s jakou fází je možné nastavit příslušnými bity v řídících registrech.

3.2.3.1 Programátorský model

Periferie se obsluhuje skrze sedm 8-bitových registrů. Ty jsou umístěny na bázové adrese MBAR + 0x0F00. Jedná se o následující registry, adresa v závorce je offset registru vzhledem k bázové adrese.

- Control register 1 (0x00)
- Control register 2 (0x01)
- Baud Rate Register (0x04)
- Status Register (0x05)
- Data Register (0x09)
- Port Data Register (0x0D)
- Data Direction Register (0x10)

Nyní se zaměřím na popis jednotlivých registrů

3.2.3.2 Control Register 1

V tomto registru se nacházejí důležité bity pro základní nastavení chování periferie

Tabulka 3.17: Control register 1

0	1	2	3	4	5	6	7
SPIE	SPE	~~SWOM~~	MSTR	CPOL	CPHA	SSOE	LSBFE
0	0	0	0	0	1	0	0

Význam bitů je následující

- *SPIE* - Příznak povolení generování přerušení pro řadič přerušení
- *SPE* - Povolení periferie
- *SWOM* - Není iplementováno
- *MSTR* - Určuje režim periferie, zda-li funguje jako master (=1) nebo jako slave (=0)

- *CPOL* - Určuje polaritu hodinové signálu
  0 - Hodiny jsou aktivní v log.1
  1 - Hodiny jsou aktivní v log.O
- *CPHA* - Určuje fázi hodinové signálu
  0 - První hrana hodinového je v polovině vysílání bitu
  1 - První hrana hodinového je souhlasně s vysílaním bitu
- *SSOE* - Povolení používání výstupu SS, tento bit má význam pouze v režimu master
- *LSBFE* - Určuje pořadí odvysílávaných bitů, pokud má hodnotu 0, je vysílan nejdříve nejvýznamější bit v opačném případě nejméně významný

3.2.3.3 Control Register 2

Registr obsahuje dodatečné bity pro nastavení periferie.

Tabulka 3.18: Control register 2

0	1	2	3	4	5	6	7
Vyhrazené						SPISWAI	SPC0
0	0	0	0	0	0	0	0

Význam bitů je následující

- SPISWAI - Bit použit pro snížení spotřeby během neaktivity, hodnota 1 znamená zastavení generátoru hodinového signálu
- SPC0 - Tento bit společně s bitem MSTR povoluje obousměrný 3-vodičový režim sběrnice

3.2.3.4 Baud Rate Register

Registr slouží pro nastavení přenosové rychlosti v režimu master.

Tabulka 3.19: Baud Rate Register

0	1	2	3	4	5	6	7
Vyhr.	SPPR			0	SPR		
0	0	0	0	0	0	0	0

Význam bitů je následující

- SPPR - Nastavení předděličky

- SPR - Výběr hodinového signálu

Frekvence hodinové signálu je odvozena z frekvence IPB, vydělením číslem vzniklým následovně
$SPI\ divisor = (SPPR + 1) \cdot 2^{SPR+1},\ SPI\ Clock = \frac{IPB\ Clock}{SPI\ Divisor}$

3.2.3.5 Status register

Tento registr poskytuje stavové informace o periferii

Tabulka 3.20: Status register

0	1	2	3	4	5	6	7
SPIF	WCOL	Vyhr.	MODF	Vyhrazené			
0	0	0	0	0	0	0	0

Význam bitů je následující

- SPIF - Indikuje konec vysílání bytu, je nastaven po osmém cyklu SCK. Vynulování se provádí přečtením tohoto registru následovaný čtením nebo zápisem z/do datového registru
- WCOL - Indikuje konflikt při zápisu do datového registru, bit je nastaven pokud dojde v průběhu přenosu k zápisu nových dat do datového registru. Vynulování se provádí stejně jako v případě bitu SPIF
- MODF - Příznak selhání, bit je nastaven pokud se na pinu /SS objeví log.0, když je periferie v režimu master

3.2.3.6 Data Register

Tento registr slouží jako datový registr, zápisem do tohoto registru dojde ke startu odvysílání tohoto bytu na sběrnici, naopak čtením získáme data přijatá.

3.2.3.7 Port Data Register, Data Direction Register

Tyto registry slouží k přímému ovládání pinů periferie, pro realizaci konfigurace FPGA tyto registry nepořebujeme, proto případné zájemce odkazuji na dokumentaci k MPC5200 firmy Freescale.

3.2.4 U-Boot

Pro implementaci konfiguračního procesu byl zvolen bootloader U-Boot, který je již na desce nahrán. V tomto bootloaderu již podpora pro FPGA existuje, je napsáno zpracování souboru s BitStreamem a vlastní algoritmus konfigurace. Je tedy nutné napsat implementaci funkcí závislých na cílové platformě. Jak jsem již dříve uvedl, U-Boot provádí konfiguraci přímým nastavováním I/O pinů, tento způsob je ale pro embedded systém pomalý. Rozhodl jsem se tedy využít dříve popsané SPI periferie a doplnit rutiny v U-Bootu tak, aby bylo možné použít bytový přístup vhodný pro SPI. Tyto změny budou popsány v následujících odstavcích.

3.2.4.1 Inicializace protředí

Pro použití FPGA v U-Bootu je nejdříve nutné bootloaderu předat informace jaké typy FPGA máme v systému k dispozici. Názvy použitých struktur dále v textu odpovídají použití FPGA firmy Xilinx Inc. a metodu slave serial, pro jiného výrobce případně jiné metody se musí názvy upravit dle příslušných hlavičkových souborů. K účelu konfigurace složí dvojice struktur *Xilinx_Spartan3_Slave_Serial_fns* a *Xilinx_desc* a funkce *fpga_add*.

Struktra *Xilinx_Spartan3_Slave_Serial_fns* definuje ukazatele na funkce závislé na cílové platformě, její definice je následující

```
1  /* Slave Serial Implementation function table */
2  typedef struct {
3    Xilinx_pre_fn pre;
4    Xilinx_pgm_fn pgm;
5    Xilinx_clk_fn clk;
6    Xilinx_init_fn  init;
7    Xilinx_done_fn  done;
8    Xilinx_wr_fn  wr;
9    Xilinx_post_fn  post;
10   int   relocated;
11   Xilinx_wr8_fn wr8;
12 } Xilinx_Spartan3_Slave_Serial_fns;
```

Význam položek je popsán v následujícím výčtu, položka relocated je pro nás nepodstatná

- *pre* - Funkce volána před začátkem konfiguračního procesu, v této funkci je možné provést případné inicializační kroky
- *pgm* - Funkce, která zajistí puls na signálu *PROG_B*
- *clk* - Funkce generující puls na hodinové signálu

- *init* - Funkce pro čtení stavu signálu *INIT_B*
- *done* - Funkce pro čtení stavu signálu *DONE_B*
- *wr* - Funkce pro nastavení hodnoty bitu na signálu *DIN*
- *post* - Funkce volaná po skončení celé konfigurace
- *wr8* - Funkce pro odeslání osmice bitů

Položka wr8 je specifická, jedná se o rozšíření U-Bootu o možnost odeslat najednou pole osmic bitů sériovou cestou a nikoliv osminásobným voláním dvojice clk/wr. Algoritmus konfigurace je rozšířen tak, aby v případě, že má tato položka jinou hodnotu než NULL použil tuto, jinak využije funkcí clk/wr. Tím že je položka přidána až nakonec struktury nedochází k ovlivnění již existujícího kódu a je tedy možné ho nechat beze změny. Toto rozšíření bylo nutné, neboť použítí SPI periferie se způsobem volání clk/wr by znamenalo v prvním případě zpomalení konfigurace a v druhém zesložitění celého kódu. Ve funkci wr by se muselo provádět bufferování předchozích sedmi bitů a s příchodem osmého je hromadně odeslat. Z těchto důvodů bylo tedy přikročeno k doplnění této položky a tím i zvýšení čitelnosti výsledného kódu.

Na našem hardware jsou komplikací signály *DONE* a *INIT_B*, tyto signály nejsou do PowerPC připojeny. Zasahovat do algoritmu a měnit ho tak, aby jejich funkce nevyužíval by znamenalo znefunkčnění ostatních implementací. Algoritmus začátku konfigurace je v U-Bootu následující. Dojde k nastavení signálu *PROG_B* do log.0 a čeká se, až přejde signál *INIT_B* také do log.0, poté se vrátí *PROG_B* zpět do log.1 a opět se čeká, až se *INIT_B* vrátí do log.1. Funkce init tedy musí splnit požadavek prvního čtení nulového a druhého čtení jedničkového, to se provede následující definicí

```
1  /*
2   * Test the state of the active-low FPGA INIT line.  Return 1 on INIT
3   * asserted (low).
4   */
5  int fpga_init_fn(int cookie)
6  {
7    static int value = 0;
8    value = 1 - value;
9    return value;
10 }
```

Na úspěšnost inicializace je spoléháno vložením dostatečných zpoždění, které zajistí korektnost konfigurace. Funkce done je testována až na konci algoritmu, kdy se čeká až vrátí hodnotu jedna. Funkce ji tedy vrací vždy.

Struktura *Xilinx_Spartan3_Slave_Serial_fns* má v našem případě inicializaci

```
Xilinx_Spartan3_Slave_Serial_fns shark_fpga_fns = {
  fpga_pre_config_fn,
  fpga_pgm_fn,
  NULL,
  fpga_init_fn,
  fpga_done_fn,
  NULL,
  fpga_post_config_fn,
  0,
  fpga_wr8_fn
};
```

Druhá struktura slouží pro definici konkrétního typu FPGA a přiřazení příslušné struktury *Xilinx_Spartan3_Slave_Serial_fns*. Struktura má definici

```
typedef struct {        /* typedef Xilinx_desc */
    Xilinx_Family    family;  /* part type */
    Xilinx_iface     iface;   /* interface type */
    size_t           size;    /* bytes of data part can accept */
    void *           iface_fns; /* interface function table */
    int              cookie;  /* implementation specific cookie */
} Xilinx_desc;          /* end, typedef Xilinx_desc */
```

Struktura se neinicializuje přímo, ale pomocí maker pro konkrétní typ FPGA. V našem případě tedy

```
Xilinx_desc fpga[1] = {
  XILINX_XC3S1400A_DESC(
    slave_serial,
    (void *)&shark_fpga_fns,
    0),
};
```

slave_serial určuje metodu konfigurace, *shark_fpga_fns* je ukazatel na předchozí strukturu naplněnou patřičními ukazateli a třetí je int hodnota, která se předává do všech funkcí a slouží k předání nějaké námi zvolené hodnoty.
Po definici struktur můžeme inicializovat FPGA framework a začlenit FPGA do systému, to provedeme voláním tří funkcí, které je ve funkci *shark_init_fpga* volané z *misc_init_r*

```
  fpga_init(gd->reloc_off);
  fpga_serialslave_init ();
  fpga_add (fpga_xilinx, &fpga[0]);
```

První dvě volání inicializují vlastní framework a třetí je vlastní začlenění FPGA. Tato funkce má dva parametry, prvním je typ (buď Xilinx nebo Altera) a druhým ukazatel na předchozí strukturu.

Funkce *shark_init_fpga* provádí dále inicializaci SPI periferie a čítače 2, kde je využito jeho funkce prostého výstupu. SPI je inicializováno následovně

- Režim master
- Hodinový signál je v klidovém stavu v log.1 (CPOL = 1), signál je fázově posunut (CPHA = 1) a jeho frekvence je 66 MHz (pro IPB = 132 MHz)
- 3-Vodičový (obousměrný) režim

Inicializace FPGA vypadá následovně

```
1  /* Configure SPI interface */
2  fpga_reg_out(&spi->brr, 0x00); // Clk = 66 MHz kHz (IPB = 132 MHz)
3  fpga_reg_out(&spi->cr1, SPI_MSTR | SPI_CPOL | SPI_CPHA);
4  fpga_reg_out(&spi->cr2, SPI_SPC0);
5  fpga_reg_out(&spi->ddr, 0x0f);
6
7  fpga_reg_out(&spi->cr1, fpga_reg_in(&spi->cr1) | SPI_SPE);
8  fpga_reg_out(&spi->pdr, 0x08);
9
10 /* Configure GPIO's - Timer2 */
11 struct mpc5xxx_gpt *gpt2 = (struct mpc5xxx_gpt*) (MPC5XXX_GPT + 0x10 *
       CFG_PROGB_TIMER);
12 gpt2->emsr = 0x34; // Used as GPIO, output, log1
```

Tímto je inicializační část hotova. Dalším krokem je odeslání vlastního BitStreamu.

3.2.4.2 Odesílání BitStreamu

Vlastní odeslání je U-Bootem řešeno pomocí smyčky, která prochází jednotlivé byty BitStreamu a volá funkce wr/clk nebo wr8. Při každém průchodu smyčkou se testuje stav signálu INIT_B, zda-li nedošlo k chybě způsobující ukončení konfigurace. V této části kódu jsem musel udělat úpravy, aby bylo možné využít SPI a blokový přenos. Hlavička pro funkci wr8 je následující

```
1 int Xilinx_wr8_fn( unsigned char *data, int length, int flush, int cookie );
```

Prvním parametrem je ukazatel na blok dat k odeslání, druhým je jejich délka, třetí je nevýznamný a čtvrtý předává hodnotu specifikovanou při deklaraci FPGA (viz 3.2.4.1). Funkce vrací skutečný počet odvysílaných dat.

Realizace vysílání znaků přes SPI je řešena pomocí přerušení. Výměna dat mezi U-Bootem a rutinou obsluhy přerušení probíhá pomocí kruhové fronty, jejíž velikost je možné měnit v mocninách dvou. Tato fronta je realizována jako lineární pole

s ukazatelem zápisu a čtení, kdy pro dosažení kruhovosti se využívá operace logického součinu hodnoty ukazatele a délky fronty, odtud tedy vyplývá požadavek na omezený výběr délky fronty. Ukazatele zápisu a čtení jsou trvale rostoucí a logický součin se provádí pouze při přístupu do pole, to je z důvodu, aby bylo vždy možné jednoznačně určit, který ukazatel je více vepředu. Může nastat případ, kdy se snažíme do bufferu zapsat více dat než je možný pojmout. S ukazateli v rozsahu od nuly do délky bufferu by nebylo možné rozlišit, zda-li je buffer ještě prázdný.

Vzhledem k velikosti BitStreamu (cca 4.5 Mb) jsem velikost této fronty zvolil na 16 kB. Při volání funkce wr8 tedy dojde k překopírování maximálně 16 kB do výstupní fronty, funkce vrátí skutečný zapsaný počet bytů a o tuto hodnoty se posune počitadlo v hlavní smyčce. Pokud je při volání wr8 fronta prázdná, dojde k vložení prvního bytu přímo do datového registru SPI, dále už je vysílání v režii obsluhy přerušení od dokončení vysílání bytu.

Rutina obsluhy přerušení je psána s ohledem na rychlost konfigurace, v případě přerušení máme procesor plně k dispozici, toho je zde využito. Při vstupu do rutiny zakážeme všechna přerušení a provedeme odvysílání 64 bytů z fronty, pokud je fronta kratší tak pochopitelně méně. To je provedeno poolingem, kdy zapíšeme byte do datového registru a čtením stavového registru čekáme až bude bit SPIF mít hodnotu 1. Po skončení vysílání zjistíme zda-li fronta není prázdná, pokud ano tak provedeme tzv. dummy čtení datového registru aby jsme vynulovali příznak SPIF. Posledním krokem je povolení přerušení, pokud fronta nebyla prázdná, dojde opět k vyvolání přerušení a cyklus se opakuje.

Použitím tohoto postupu vysílání se podařilo dosáhnout času konfigurace 0.6 s, zatímco v případě vysílání pouze jednoho znaku v obsluze přerušení konfigurace zabrala přibližně 10 s. Tento rozdíl je způsoben plánováním úloh U-Boota, kdy hlavní smyčka je vykonávána s taktem 1 ms. Postup řešení s vícenásobným vysíláním v obsluze přerušení by se na první pohled mohl zdát jako porušení zásad psání těchto rutin, kdy rutina by měla být co nejkratší. Pokud však přihlédnu k předpkládanému využití, kdy během konfigurace FPGA není potřeba provádět jinou činnost natož interakci s uživatelem, jedná se dle mého názoru o vhodné řešení.

Posledním krokem po odvysílání BitStreamu je vygenerování dostatečného počtu hodinových cyklů pro korektní start FPGA, toho je dosaženo vysíláním bytů s hodnotou 0xFF po definovanou dobu. Tímto posledním krokem je konfigurace dokončena a FPGA je připraveno k použití. Výpis zdrojových kódu s úpravami v U-Bootu je k dispozici na přiloženém CD.

Kapitola 4

Piranha, PikeOS

4.1 Popis

PikeOS ([4]) je platforma pro vývoj embedded zařízení, kde může běžet více operačních systémů a aplikací simultáně v odděleném a robustním prostředí. Architektura PikeOS je založena na mikrokernelu, které provádí virtualizaci procesoru a některých perfierií a poskytuje minimální skupinu služeb. Cílem této kapitoly je vytvořit demostrační aplikaci na platformě PikeOS, která bude zpracovávat data z modulů na sběrnici PIRANHA a umožní jejich diagnostiku.

4.2 Inicializace sběrnice PIRANHA

Inicializací sběrnice se rozumí nastavení režimu mastera a definice modulů, přítomných na sběrnici. Pro tento účel slouží služební oblast paměti FPGA, která je U-Bootem namapována na fyzickou adresu 0xFB000000. Služební RAM je rozdělena na 64 stejných bloků každý o velikosti 64 byte, jde o jeden blok pro konfiguraci mastera, šedesát dva bloků pro moduly a jeden pro služební FIFO.

Inicializací se tedy rozumí naplnění těchto bloků definicí připojených modulů. Po této incializaci jsou již v paměti FPGA k dispozici přečtená data ze vstupů v části DATA-IN (offset oproti bázové adrese 0x1000) a zároveň je možné zápisem do části DATA-OUT (offset oproti bázové adrese 0x1200) ovládat výstupní moduly.

4.2.1 Konfigurace mastera

Struktura bloku mastera je v tabulce 4.1, offset bloku je 0x0000.

Tabulka 4.1: Seznam registrů - Master

	0	1	2	3	4	5	6	7
0x00	Status	MStatus	MCMA	MCMB	MCMC	MCMD	Typ	Typ
0x08		FIFOFULL	FIFOEMPTY	FIFOSET	MDIAG	MDIAG	TIMEE	TIMEE
0x10	TIMEC		TIMER1a	TIMER1b	TIMER1c	TIMER1d	TIMER2a	TIMER2b
0x18	Sumes	Sumes	Sumes	Sumes	ErrMes	ErrMes	ErrMes	ErrMes
0x20	MTIMEA	MTIMEA	NEXTADRa	NEXTADRb	Change		Timeout	Timeout
0x28		ctderemax	nulmax	CtDotFoutMax	CtOdpFinMax	CekejDotMax		
0x30	NxtTmSt01	NxtTmSt01	NxtTmSt02	NxtTmSt02	NxtTmSt03	NxtTmSt03	NxtTmSt04	NxtTmSt04
0x38								

V tabulce 4.2 je uveden význam vybraných registrů

Tabulka 4.2: Popis registrů - Master

Název	Offset	Význam
Status	0x00	Řídící registr mastera
MStatus	0x01	Stavový registr
MCMA	0x02	Adresa modulu, pro který se provádí speciální režim
MCMB	0x03	Příkaz, který se provádí ve speciálním režimu
MCMD	0x05	Interval mezi časy, kdy master cyklicky kontroluje registr STATUS
Typ	0x06, 0x07	Typ mastera
MDIAG	0x0C, 0x0D	Diagnostická informace mastera
SUMES	0x18 - 0x1B	Počet správně odkomunikovaných zpráv
ERRMES	0x1C - 0x1F	Počet chybných zpráv
ctderemax	0x29	Interval použitý k ustálení linky po přechodu do třetího stavu

Všechny registry v služební RAM jsou 8-bitové a v následujících podkapitolách následuje popis jednotlivých bitů těchto registrů

4.2.1.1 Status

Tento registr slouží pro povolení řízení činností mastera a úrovně rozdělení činností mezi mastera a PowerPC.

Tabulka 4.3: Popis registru STATUS

Bit	Název	Význam
0	DataCycle	Povoluje datový režim - provádí se čtení a zápis dat z/do modulů
1	Mode	Povoluje speciální režim - konfigurační zprávy
2,3	Autonomie	00 - žádná úroveň autonomie, vše řízeno nadřazeným členem 01 - master v rámci některých režimů mění STX 10 - master provádí detekci modulů 11 - maxiální úroveň autonomie, zatím nevyužito
4	AllowI	Povoluje masterovi přístup do RAM-IN
5	AllowO	Povoluje masterovi přístup do RAM-OUT
6	Nevyužito	
7	Nevyužito	

Pro využití speciálního režimu se musí dodržet následující postup

- Nejdříve nastavíme registr MCMA na adresu modulu, se kterým chceme provádět speciální obsluhu
- V registru MCMB nastavíme jaká akce se má s modulem provádět
- Nastavíme bit Mode v registru STATUS
- V bloku slave nastavíme nejvyšší bit STATUS registru PROCESS, to zajistí zpracování bitu Mode v STATUS registru a tím dojde k provedení speciální obsluhy

4.2.1.2 MStatus

Registr slouží pro získání stavových informací z mastera.

Tabulka 4.4: Popis registru MSTATUS

Bit	Název	Význam
0	GO	Pokud má hodnotu 1, master je v činnosti
1	SGO	0 - provádí se cykly přenosu dat 1 - provádí se speciální obsluha
2	Nevyužito	
3	RESET	Indikace, že je master připraven k činnosti
4	RequestI	Master pracuje s pamětí RAM-IN
5	RequestO	Master pracuje s pamětí RAM-OUT
6	Nevyužito	
7	Err	Chyba mastera

4.2.1.3 MCMB

V tabulce 4.5 je uveden seznam příkazů speciální obsluhy modulů, příkaz je určen spodními 6 bity tohoto registru

Tabulka 4.5: Definice příkazů v registru MCMB

Příkaz	Význam
0x00	Žadný příkaz
0x01	Po resetu provede kontrolu systému po diagnostické lince a přiřadí adresy
0x02	Po resetu během činnosti provede kontrolu systému po komunikační lince a obnovení nastavení modulu z Flash
0x03	Po resetu provede kontrolu systému po komunikační lince a obnoví nastavení modulu z Flash
0x07	Kontrola systému pomocí jednoho dotazu a hromadné odpovědi
0x08	Zjištění časové prodlevy u jednotlivých modulů, použito pro přesnou synchronizaci pod 1us
0x09	Dotaz na diagnostiku modulu
0x0A	Přepnutí diagnostické linky na přenos dat

Příkaz	Význam
0x0D	Synchronizace na vnější událost
0x10	Uložení adresy do Flash
0x11	Uložení nastavení do Flash
0x13	Čtení a kontrola obsahu Flash a všech registrů modulu

4.2.2 Konfigurace modulů

Struktura bloku modulů je v tabulce 4.6, offset bloku je 0x0000 + n * 0x40, kde n je číslo modulu od 1 do 62.

Tabulka 4.6: Seznam registrů - Modul

	0	1	2	3	4	5	6	7
0x00	STX	CMA	CMB	CMC	CMD	ETX	TypA	TypB
0x08	BaseInA	BaseInB	BaseOutA	BaseOutB	DIAG	DIAG	Time	Time
0x10	TimeC	TimeDA	TimeDB	TimeDC	TimeLA	TimeLB	PROCES	OrgProc
0x18	SumesA	SumesB	SumesC	SumesD	ErrMes	ErrMes	ErrMes	ErrMes
0x20	TIMEA	TIMEA	NEXTADRa	NEXTADRb			Timeout	Timeout
0x28	Addr	ctderemax	nulmax	In *	BackAdrA	BackAdrB	Out *	ErrM
0x30	NextTime	NextTime		CFRAM	AdrFram	AdrFram	FRAM0	FRAM1
0x38	FRAM2	FRAM3	FRAM4	FRAM5	FRAM6	FRAM7		

V tabulce 4.7 je uveden význam vybraných registrů

Tabulka 4.7: Popis registrů - Master

Název	Offset	Význam
STX	0x00	Adresa modulu
CMA	0x01	Nastavení režimu diagnostické linky a zabezpečení
CMB	0x02	Čtení diagnostiky a Flash, ukládání nastavení
CMC	0x03	Resetování, Testovací odpovědi
CMD	0x04	Nastavení přenosové rychlosti a formátu dat
ETX	0x05	Obsahuje časté řídící bity, řídí synchronizaci více modulů
Typ	0x06, 0x07	Typ mastera

Název	Offset	Význam
BaseIn	0x08, 0x09	Adresa v RAM-IN kam se mají ukládat načtená data z modulu
BaseOut	0x0A, 0x0B	Adresa v RAM-OUT odkud se budou zapisovat data do modulu
DIAG	0x0C, 0x0D	Vyčtená základní diagnostika modulu
Time, TimeC	0x0E - 0x10	Čas po kterém musí být modul dotazován masterem na nová data nebo nová data zapsána
PROCESS	0x16	Příznaky změny, určují speciální operaci s modulem mimo datový cyklus
TimeD	0x11 - 0x13	Zpoždění, po kterém má modul odpovídat po přijetí hromadného dotazu. Definováno v počtu 20ns tiků
TimeL	0x14, 0x15	Zpoždění pro synchronizaci, využito při vzdálených modulech, kdy se lokální moduly pomocí tohoto nastavení zpomalují. Definováno v počtu 20ns tiků
In	0x1B	Počet vstupních dat modulu, udáno v bytech
Out	0x1E	Počet výstupních dat modulu, udáno v bytech
TIMEA	0x20, 0x21	Čítač aktuálního času od poslední obsluhy modulu
ctderemax	0x29	Interval použitý k ustálení linky po přechodu do třetího stavu

4.2.2.1 CMA

Jedná se o trvalé řídící slovo. Jeho nastavení ovlivňuje, jak modul pracuje s diagnostickou linkou a zda se rámce opatřují kontrolním součtem. V registru jsou využity pouze bity 4 až 0, ostatní bity musí mít při zápisu hodnotu nula.

- *bit4 - cma4* - Určuje zda modul bude odpovídat na hromadný dotaz
- *bit3 - cma3* - Režim diagnostické linky, 1..diagnostika, 0..data
- *bit2 - cma2* - Směr přenosu po diagnostické lince, 1..Vysílání, 0..Příjem
- *bit1 - cma1* - Řídí bypass diagnostické linky, 1..aktivní, 0..nekativní
- *bit0 - cma0* - Za odesílané zprávy připojovat kontrolní byte

4.2.2.2 CMB

Jedná se o řídící slovo, které se uplatňuje pro zprávu s ním odesílanou. Jeho využití je pro čtení diagnostických informací a práce s Flash pamětí modulu. V registru jsou využity bity 4 až 0, ostatní bity musí mít při zápisu hodnotu nula.

- *bit4,bit3 - cmb4,cmb3* - Určuje zda modul bude odpovídat na hromadný dotaz

CMB4	CMB3	Význam
0	0	Žádný dotaz na diagnostiku
0	1	Dotaz na krátkou diagnostiku - 2 byty
1	0	Dotaz na dlouhou diagnostiku
1	1	Vyčtení všech registrů včetně dat

- *bit2,bit1,bit0 - cmb2,cmb1,cmb0*

CMB2	CMB1	CMB0	Význam
0	0	0	Zatím nevyužito
0	0	1	Nastavení adresy, za CMDB následuje byte s adresou a hodnotou ctderemax
0	1	0	Následuje nastavení parametrů TIMED a TIMEL
0	1	1	Následuje adresa bloku dat
1	0	0	Zatím nevyužito
1	0	1	Následují nastavovací hodnoty podle typu modulu
1	1	0	Zatím nevyužito
1	1	1	Zatím nevyužito

4.2.2.3 CMC

Jedná se o řídící slovo, které se uplatňuje pro zprávu s ním odesílanou. Jeho využití je pro testovací účely. V registru jsou využity bity 4 až 0, ostatní bity musí mít při zápisu hodnotu nula.

- *bit4 - cmc4* - Vynulování posuvných registrů diagnostické linky
- *bit3 - cmc3* - Uloží všechny parametry modulu do Flash
- *bit2 - cmc2* - Načte adresu modulu z Flash a provede výchozí nastavení
- *bit1 - cmc1* - Vynuluje adresu modulu
- *bit0 - cmc0* - Reset všech registrů modulu a provedení výchozího nastavení

4.2.2.4 CMD

Toto řídící slovo je trvalé a slouží k nastavení přenosové rychlosti a formátu předávaných dat. V registru jsou využity bity 4 až 0, ostatní bity musí mít při zápisu hodnotu nula.

- *bit4 - cmd4* - Pokud má hodnotu 0, jsou vysílána data ve fromátu “Data0 Data1 ... DataN”, pokud má hodnotu 1, jsou vysílána data ve formátu “AdrdatX DataX AdrdatY DataY ...”
- *bit3,bit2,bit1,bit0 - cmd3,cmd2,cmd1,cmd0* - Určuje přenosovu rychlost, jde o počet 10ns přírůstků k základní periodě hodinové signálu 50ns

4.2.2.5 ETX

Jedná se o řídící slovo, které obsahuje bity potřebné pro společnou synchronizaci více modulů. Toho se využívá pro synchronizované nastavení výstupu a přečtení vstupů

- *bit5 - etx5* - Provede se synchronizace modulů, které dříve obdržely zprávu s nastaveným etx4
- *bit4 - etx4* - Informuje modul, že má čekat na synchronizaci
- *bit3 - etx3* - Určuje směr přenosu na komunikační lince, využito pro opakovače. Nedoporučuje se měnit.
- *bit2 - etx2* - Směr přenosu na diagnostické lince. Nedoporučuje se měnit.
- *bit1 - etx1* - Informuje modul, že má potvrzovat přijetí odpovědi
- *bit0 - etx0* - Určuje, že je vysíláno CRC u rámců

4.2.2.6 Typ

Tato dvojice registrů definuje typ připojeného modulu a určuje, zda-li se jedná vstupní, výstupní nebo kombinovaný modul. Bity 15 a 14 definují směr signálů modulů následovně

Tabulka 4.8: Směr signálů daný typem modulu

bit15	bit14	Význam
0	0	Nedatový modul (např. Opakovač)
0	1	Vstupní modul
1	0	Výstupní modul
1	1	Kombinovaný modul

Bity 5 až 0 registru určují konkrétní variantu modulu.

4.2.2.7 Process

Tento registr popisuje režim chování modulu v rámci celého systému. Význam jednotlivých bitů je následující

- *bit7 - STATUS* - Pokud má hodnotu jedna, určuje, že se bude po zpracování modulu testovat bit Mode v registru STATUS mastera
- *bit6 - bit2* - Režim činnosti, význam je uveden v tabulce 4.9
- *bit1 - CMx* - Pokud má hodnotu jedna, provedou se změny zapsané v regitrech CMx
- *bit0 - Obsluz* - Pokud má hodnotu nula, neprovádí se oblsuha modulu

Podporované režimy činnosti

Tabulka 4.9: Význam bitů 6 až 2 v registru PROCESS

bit6-2	Význam
00000	Nemá význam
00001	Modul není ve skupině, Má vlastní TIME, provádí se kontrola TIMEA. Data se zpracovávají ihned po požadavku
00010	První modul nesynchronizované skupiny. Má stejné TIME jako následující modul. Kontrola TIMEA se provádí. Data se přepisují na výstup a čtou vstupní hned po dotazu.
00011	Modul je v nesynchronizované skupině. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se přepisují na výstup a čtou vstupní hned po dotazu.

00100	Poslední modul nesynchronizované skupiny. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se přepisují na výstup a čtou vstupní hned po dotazu.
00101	První modul synchronizované skupiny. Má stejné TIME jako následující modul. Kontrola TIMEA se provádí. Data se přepisují na výstup a čtou vstupní až po synchronizačním rámci po posledním dotazu.
00110	Modul je v synchronizované skupině. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se přepisují na výstup a čtou vstupní až po synchronizačním rámci po posledním dotazu.
00111	Poslední modul synchronizované skupiny. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se přepisují na výstup a čtou vstupní až po synchronizačním rámci po posledním dotazu. Po této zprávě master musí vygenerovat hromadnou synchronizační zprávu. Po té se přepíší zapsané výstupní data na výstup a sejmou se. Čtou se v rámci dalšího cyklu po jednotlivých zprávách.
01000	Poslední modul synchronizované skupiny. Nesynchronizuj následující zprávou ale po řízení z PowerPC
01001	První modul nesynchronizované skupiny s hromadným čtením dat. Má stejné TIME jako následující modul. Kontrola TIMEA se provádí. Data se zapisují po přijetí výstupní zprávy. Data se čtou po posledním dotazu skupiny hromadným dotazem, po kterém se očekávají data od všech modulů, které mají nastaveny cma4 (týká se i jiných skupin).
01010	Modul je v nesynchronizované skupině s hromadným čtením dat. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se zapisují po přijetí výstupní zprávy. Data se čtou po posledním dotazu skupiny hromadným dotazem, po kterém se očekávají data od všech modulů, které mají nastaveny cma4 (týka se i jiných skupin).
01011	Poslední modul nesynchronizované skupiny s hromadným čtením dat. Má stejné TIME jako předchozí modul. Kontrola TIMEA se neprovádí. Data se zapisují po přijetí výstupní zprávy. Data se čtou po posledním dotazu skupiny hromadným dotazem po kterém se očekávají data od všech modulů, které mají nastaveny cma4 (týka se i jiných skupin). Po této zprávě mastera musí vygenerovat hromadnou zprávu s dotazem na data. Po té začnou přicházet jednotlivé zprávy podle zpoždění TIMED od všech vstupních modulů s nastaveným cma4.

01100	První modul synchronizované skupiny s hromadným vyčtením. Má stejné TIME jako následující modul. Kontrola TIMEA se provádí. Data se přepisují na výstup a čtou vstupní až po synchronizačním rámci po posledním dotazu. Data se čtou po synchronizačním rámci hromadným dotazem, po kterém se očekávají data od všech modulů, které mají nastaveny cma4 (týka se i jiných skupin).
01101	Modul je v synchronizované skupině s hromadným vyčtením.
01110	Poslední modul synchronizované skupiny s hromadným vyčtením. Synchronizuj a spusť hromadné vyčtení dat.
01111	Poslední modul synchronizované skupiny s hromadným vyčtením. Nesynchronizuj a nespuštěj hromadné vyčtení dat.
11110	Zjišťovaní přítomnosti modulu po čase TIMEE. Po vypršení TIMEE se po diagnostické lince adresuje chybějící modul a zjišťuje se nejdříve adresa. Pokud odpoví následující adresou sousedniho modulu je jasné, že chybí. Chybět může více modulů.

4.3 Přístup k datům

V této části bude popsán algoritmus čtení a zápisu do společné části RAM-IN a RAM-OUT ze strany PowerPC. Tyto dvě části paměti jsou používány jak PowerPC tak i masterem, hrozí tak situace, kdy jedna ze stran může přečíst nekompletní data. V případě čtení například čtyř bytů z modulu může dojít k situaci, kdy jedna strana přečte dva byty nové a dva byty z předchozího čtení. Výsledkem jsou poté nekonzistentní data. Z tohoto důvodu bylo nutné implementovat algirtmus, který těmto případům předejde.

Navržený algoritmus spočívá v zamezení přístupu mastera ke sdíleným pamětem. Pro činnost algoritmu jsou využity bity *AllowI* a *AllowO* v STATUS registru a bity *RequestI* a *RequestO* v MSTATUS registru. Pokud chce PowerPC získat přístup k bloku paměti nastaví odpovídající bit Allow* do nuly, poté čeká až bude bit Request* nulový. Jakmile je tato podmínka splněna, počká PowerPC definovanou dobu a opětovně přečte bit Request*. Pokud je i tentokrát nulový, má PowerPC pamět k dispozici. Po čtení nebo zápisu bloku PowerPC vrátí bit Allow* zpět do jedničky, tímto je operace ukončena.

4.4 Aplikace v PikeOS

Realizovaná aplikace je napsána v jazyce C++ na platformě PikeOS. Obsahuje funkce pro manipulaci s definicí modulů v služební RAM a provádí cyklické vyčítání obsahu paměti RAM-IN a její obsah zapisuje do RAM-OUT. Provádí tedy kopírování vstupů na výstupy. Zároveň na konzoli vypisuje diagnostické informace.

Pro ovládání mastera byla napsána třída, která zapouzdřuje sady registrů a usnadňuje jejich konfiguraci. Dále se zaměřím na popis metod této třídy a poté na její použití.

4.4.1 Programový modul Piranha

Tento modul obsahuje definici struktury pro blok mastera, strukturu pro slave moduly a třídu Piranha pro přístup k modulům. Definice struktur přesně odpovídá definicím příslušných bloků v služební RAM a tím slouží ke zjednodušní přístupu, jejich definice je v příloze B. Následuje popis jednotlivých metod třídy Piranha.

4.4.1.1 Konstruktor

Konstruktor třídy Piranha má definici

```
Piranha(char *baseAddressPath);
```

Konstruktor provede získání bázové adresy mastera z property filesystému a provede namapování mastera do paměťového prostoru běžící aplikace. Využitím property filesystému můžeme bázovou adresu měnit bez nutnosti zasahovat do zdrojových kódů. Vytvořením instance třídy je vše připraveno pro využití následných metod.

Význam parametrů

- *baseAddressPath* - Udává cestu v property filesystému PikeOS, kde se nachází bázová adresa mastera v paměti (v mém konkrétním případě je adresa 0xFB000000 viz. 3.1.5)

4.4.1.2 GetMasterStatus

Metoda slouží pro získání hodnoty stavových registrů *STATUS* a *MSTATUS*.

```
short int GetMasterStatus();
```

Data jsou vráceny jako 16-bitová hodnota, kde vyšší byte odpovídá registru STATUS a nižší byte registru MSTATUS.

4.4.1.3 Metody pro konfiguraci mastera

Tyto metody slouží pro konfiguraci chování mastera. Jedná se o následující tři metody

```
void SetControlAndStatus(char controlAndStatus);
void SetMasterCMx(char MCMA, char MCMB, char MCMC, char MCMD);
void SetMasterCtDEREMax(char value);
```

Metoda *SetControlAndStatus* slouží pro manipulaci s registrem STATUS mastera. Význam parametrů

- *controlAndStatus* - Hodnota zapisovaná do registru STATUS

Metoda *SetMasterCMx* slouží k počátečnímu nastavení registrů MCMA, MCMB, MCMC a MCMD. Parametry metody přímo určují obsahy těchto registrů

Metody *SetMasterCtDEREMax* nastavuje ochrannou dobu, pro přepnutí sběrnice do třetího stav. Parametrem metody je přímo hodnota registru CtDEREMax mastera.

4.4.1.4 Metody pro konfiguraci modulů

Tyto metody slouží pro definici modulů připojených do systému. Jedná se o následující metody

```
void ClearModule(int position);
void AssignAddress(int position, char address);
void SetModuleCMx(int position, char CMA, char CMB, char CMC, char CMD);
void SetModuleETX(int position, char ETX);
void SetModuleType(int position, int moduleType);
void SetCommunicationDelays(int position, char timeC, int timeD, int timeL);
void SetProcess(int position, char process);
void SetModulePositionInCommList(int position, int nextModuleIndex, int
    prevModuleIndex);
void SetRAMRegions(int position, int inAddress, char inCount, int outAddress, char
     outCount);
```

Každá z těchto metod má jako první parametr číslo pozice, kterou konfigurujeme. Ta je v intervalu od jedné do šedesáti dvou včetně a určuje umístění modulu na společné liště. Parametry metod přímo odpovídají obsahu příslušných registrů.

Metoda *ClearModule* slouží pro vynulování oblasti patřící dané pozici a tím uvede pozici do známého stavu.

Metoda *AssignAddress* nastavuje adresu modulu, který se nachází v dané pozici. Jde o nastavení registru STX a Addr. Adresa podléha podmínkám adresování sběrnice Piranha, je tedy v rozsahu od jedné do šedesáti dvou.

Metoda *SetModuleCMx* slouží k nastavení registrů CMA, CMB, CMC a CMD.

Metoda *SetModuleETX* slouží k nastavení registru ETX.

Metoda *SetModuleType* slouží k nastavení typu modulu.

Metoda *SetCommunicationDelays* nastavuje doby zpoždění při hromadném dotazu (TimeD) a pro synchronizaci (TimeL)

Metoda *SetProcess* slouži pro nastavení režimu činnosti modulu, význam parametru je dán popisem registru *PROCESS* v sekci 4.2.2.7

Metoda *SetModulePositionInCommList* nastavuje pořadí komunikace modulu v rámci komunikační smyčky. Nastavuje se index předchozího a následujícícho modulu, indexy jsou počítány od nuly. První modul má tedy index nula.

Metoda *SetRAMRegions* nastavuje umístění vstupních a výstupních dat v pamětech RAM-IN a RAM-OUT. Určuje se offset (inAddress a outAddress) od počátku dané oblasti a počet bytů (inCount a outCount), který se čte nebo zapisuje do modulu.

4.4.1.5 Metody pro přístup k datovým oblastem

Tyto metody slouží pro přístup k datovým oblastem a to RAM-IN a RAM-OUT. Jedná se o následující metody

```
bool LockRegion(bool outRegion);
bool UnlockRegion(bool outRegion);
void * GetModuleInput(int position);
void * GetModuleOutput(int position);
```

Metody *LockRegion* a *UnlockRegion* slouží pro uzamknutí RAM-IN nebo RAM-OUT pro výhradní přístup PowerPC. Mají jeden parametr určující, který region zamykáme. Pokud má hodnotu *false* jedná se o RAM-IN, pokud hodnotu *true* pak se jedná o RAM-OUT. Metody vrací úspěšnost požadované akce, pokud metoda *LockRegion* vrátí *false*, nesmí PowerPC s pamětmi pracovat. Jinak by hrozilo by čtení nebo zápis nekonzistentních dat.

Metody *GetModuleInput* a *GetModuleOutput* vrací adresu, kde se nachází vstupní respektive výstupní data pro požadovanou pozici. Na těchto adresách se již nacházejí přímo data přečtená z modulů nebo určená k zápisu do modulů.

4.4.2 Aplikace

Díky realizaci mastera v FPGA a třídy Piranha se aplikace, která chce využívat služeb vstupně/výstupních modulů, výrazně zjednodušuje. Prvním krokem je vytvoření instance třídy Piranha se správnou cestou k bázové adrese

```
#define PIRANHA_PROPFS_PATH "/prop/app/posix/posix/piranha/mem"
Master = new Piranha(PIRANHA_PROPFS_PATH);
```

Pro další využití se musí zkontrolovat stav mastera, zda-li je připraven. To provedeme voláním metody *GetMasterStatus* a kontrolou nastaveného bitu RESET. Pokud je nastaven, může se pokračovat s definováním přítomných modulů. Na sběrnici máme k dispozici jako první modul 8-bitový výstupní a jako druhý 8-bitový vstupní. Definice tedy bude vypadat následovně

```
Master->ClearModule(1);
Master->ClearModule(2);

Master->AssignAddress(1, 1);
Master->SetModuleCMx(1, 0, 0, 0, 0);
Master->SetModuleETX(1, 0x02);
Master->SetModuleType(1, 0x8002);
Master->SetCommunicationDelays(1, 0, 0x1ff, 0);
Master->SetProcess(1, 0x5);
Master->SetModulePositionInCommList(1, 1, 1);
Master->SetRAMRegions(1, 0, 0, 0, 1);

Master->AssignAddress(2, 2);
Master->SetModuleCMx(2, 0, 0, 0, 0);
Master->SetModuleType(2, 0x4002);
Master->SetCommunicationDelays(2, 0, 0x1ff, 0);
Master->SetProcess(2, 0x5);
Master->SetModulePositionInCommList(2, 0, 0);
Master->SetRAMRegions(2, 0, 1, 0, 0);
```

Moduly se budou zpracovávat v následujícím pořadí, nejdříve výstupní a poté vstupní modul.

Dalším krokem je inicializace mastera, čili nastavení jeho režimu a provedení detekce modulů. Toto zajistí následující volání metod

```
Master->SetMasterCMx(0, 1, 0, 0);
Master->SetMasterCtDEREMax(0);
Master->SetControlAndStatus(0x32);
usleep(100000);
Master->SetControlAndStatus(0x31);
```

První dvě volání inicializují mastera, třetí spustí detekci a konfiguraci nadefinovaných modulů. Poslední spouští vlastní výměnu dat mezi masterem a definovanými

moduly.

Posledním části aplikace je již vlastní předávání dat ze vstupní na výstupní modul. Tyto akce provádí následující blok kódu

```
char *input = (char*)Master->GetModuleInput(2);
char *output = (char*)Master->GetModuleOutput(1);

while (!stop) {
  if(Master->LockRegion(false))
  {
    char Inputs = *input;
    Master->UnlockRegion(false);

    if(Master->LockRegion(true))
    {
      *output = Inputs;
      Master->UnlockRegion(true);
    }
  }
}
```

Prvním krokem je uzamčení paměti RAM-IN, pokud se tato akce zdaří, přečte se stav vstupů ze vstupního modulu a provede následné odemčení paměti. Poté následuje nastavení výstupu, opět s podmínkou zamykání paměti, tentokrát ale RAM-OUT. Po úspěšném uzamčení se přepíše hodnota vstupu na výstup a provede odemčení paměti. Tyto operace se provádí cyklicky ve smyčce až do doby ukončení aplikace. Při ukončení aplikace se provede již jen odalokování instance třídy Piranha a aplikace končí.

Kapitola 5

Závěr

Cílem této práce bylo připojení FPGA k vnějšímu paměťovému subsystému procesoru MPC5200, realizovat konfiguraci tohoto FPGA prostřednictvím bootloaderu U-Boot a napsat aplikaci na platformě PikeOS, která realizuje komunikaci na sběrnici PIRANHA. Tyto úkoly byly úspěšně realizovány, tak jak je popsáno v předchozím textu.

V této práci nebylo využito diagnostických zpráv modulů, jelikož tato funkcionalita nebyla v době psaní této práce masterem a slave moduly podporována. Realizovaná třída Piranha je však na doplnění této funkcionality připravena a tudíž v budoucnu nebude složité tuto funkcionalitu implementovat. Implementace diagnostiky modulů je ale také závislá na způsobu začlenění do cílové aplikace. Předpokládá se využití jako vstupy/výstupy budoucí aplikace realizující Programovatelný kontrolér, který musí diagnostické zprávy nějakým způsobem ukládat a v případe požadavku předat do nadřazené vrstvy.

Literatura

[1] Freescale, **Dokumentace k MPC5200**
http://www.freescale.com

[2] Mikroklima, **Embedded modul SHARK**
http://www.mikroklima.cz

[3] Univerzální bootloader U-Boot
http://www.denx.de/wiki/UBoot

[4] Sysgo AG - Realtime Solutions **PikeOS**
http://www.sysgo.com

[5] Xilinx Inc. **Katalogové listy a aplikační poznámky**
http://www.xilinx.com

Příloha A

Konfigurace FPGA

Zde je uveden realizovaný platformě závislý kód pro konfiguraci FPGA v U-Boot

A.1 fpga.c

```
/*
 * (C) Copyright 2007
 * Matthias Fuchs, esd gmbh germany, matthias.fuchs@esd-electronics.com
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

extern int shark_init_fpga(void);

extern int fpga_pgm_fn(int assert_pgm, int flush, int cookie);
extern int fpga_init_fn(int cookie);
extern int fpga_done_fn(int cookie);
```

```
extern int fpga_wr8_fn(unsigned char *value, int length, int flush, int cookie);
extern int fpga_pre_config_fn(int cookie );
extern int fpga_post_config_fn(int cookie );
```

A.2 fpga.c

```
/*
 * (C) Copyright 2007
 * Matthias Fuchs, esd gmbh, matthias.fuchs@esd-electronics.com.
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#include <common.h>
#include <asm/io.h>
#include <spartan3.h>
#include <command.h>
#include "mpc5xxx.h"
#include "fpga.h"

DECLARE_GLOBAL_DATA_PTR;

#if defined(CONFIG_FPGA)

#define CFG_PROGB_TIMER    2

Xilinx_Spartan3_Slave_Serial_fns shark_fpga_fns = {
  fpga_pre_config_fn,
  fpga_pgm_fn,
  NULL,
  fpga_init_fn,
  fpga_done_fn,
```

```
43    NULL,
44    fpga_post_config_fn,
45    0,
46    fpga_wr8_fn
47  };
48
49  Xilinx_desc fpga[1] = {
50    XILINX_XC3S1400A_DESC(
51      slave_serial,
52      (void *)&shark_fpga_fns,
53      0),
54  };
55
56  #define XMIT_BUFFER_SIZE  16383 // Size of xmit buffer MINUS ONE, !!! HAVE TO BE
        2^n-1 !!!
57  const int irq = MPC5XXX_SPI_SPIF_IRQ;
58  static struct mpc5xxx_spi *spi = (struct mpc5xxx_spi*) MPC5XXX_SPI;
59  unsigned char xmit_queue[XMIT_BUFFER_SIZE + 1];
60  static int xmit_write = 0;
61  static int xmit_read = 0;
62
63  static int fpga_reg_in(volatile u8 *reg)
64  {
65    int ret = readb(reg);
66    __asm__ __volatile__ ("eieio");
67    return ret;
68  }
69
70  static void fpga_reg_out(volatile u8 *reg, int val)
71  {
72    writeb(val & 0xff, reg);
73    __asm__ __volatile__ ("eieio");
74
75    return;
76  }
77
78  static void spi_irq_handler(void *arg)
79  {
80    int counter = 64 - 1; // -1 beacuse of loops
81
82    /* Disable SPIE interrupt */
83    disable_interrupts();
84    /* Clear IRQ flag */
85    volatile int dummy = fpga_reg_in(&spi->sr);
86
87    /* if transmit queue contains data write it to data register */
88    while( (xmit_read < xmit_write) && (counter >= 0) ) {
89      fpga_reg_out(&spi->dr, xmit_queue[(xmit_read++) & XMIT_BUFFER_SIZE]);
90      while( (fpga_reg_in(&spi->sr) & SPI_SPIF) == 0);
91      counter--;
92    }
93
94    /* If all was sent dummy read data register to clear IRQ flag */
```

```
95    if(xmit_read == xmit_write)
96      dummy = fpga_reg_in(&spi->dr);
97
98    enable_interrupts();
99  }
100
101 /*
102  * Set the active-low FPGA reset signal.
103  */
104 void fpga_reset(int assert)
105 {
106   /* DUMMY in our implementation */
107
108   debug("%s:%d: RESET ", __FUNCTION__, __LINE__);
109   if (assert) {
110     debug("asserted\n");
111   } else {
112     debug("deasserted\n");
113   }
114 }
115
116
117 /*
118  * Initialize the SelectMap interface.  We assume that the mode and the
119  * initial state of all of the port pins have already been set!
120  */
121 void fpga_serialslave_init(void)
122 {
123   debug("%s:%d: Initialize serial slave interface\n", __FUNCTION__,
124         __LINE__);
125   fpga_pgm_fn(FALSE, FALSE, 0); /* make sure program pin is inactive */
126 }
127
128
129 /*
130  * Set the FPGA's active-low SelectMap program line to the specified level
131  */
132 int fpga_pgm_fn(int assert, int flush, int cookie)
133 {
134   debug("%s:%d: FPGA PROGRAM ",
135         __FUNCTION__, __LINE__);
136
137   struct mpc5xxx_gpt *gpt2 = (struct mpc5xxx_gpt*) (MPC5XXX_GPT + 0x10 *
        CFG_PROGB_TIMER);
138   assert = 1 - assert;
139   gpt2->emsr = (gpt2->emsr & 0x2f) | (assert << 4);
140   debug("%d\n", assert);
141
142   return assert;
143 }
144
145
146 /*
```

```
147  * Test the state of the active-low FPGA INIT line.  Return 1 on INIT
148  * asserted (low).
149  */
150 int fpga_init_fn(int cookie)
151 {
152 /*  if (in_be32((void*)GPIO1_IR) & GPIO1_FPGA_INIT)
153     return 0;
154   else
155     return 1;*/
156   static int value = 0;
157   value = 1 - value;
158   return value;
159 }
160
161
162 /*
163  * Test the state of the active-high FPGA DONE pin
164  */
165 int fpga_done_fn(int cookie)
166 {
167 /*  if (in_be32((void*)GPIO1_IR) & GPIO1_FPGA_DONE)
168     return 1;
169   else
170     return 0;*/
171   return 1;
172 }
173
174
175 /*
176  * FPGA pre-configuration function. Just make sure that
177  * FPGA reset is asserted to keep the FPGA from starting up after
178  * configuration.
179  */
180 int fpga_pre_config_fn(int cookie)
181 {
182   debug("%s:%d: FPGA pre-configuration\n", __FUNCTION__, __LINE__);
183
184   xmit_write = 0;
185   xmit_read = 0;
186
187   fpga_reg_out(&spi->cr1, fpga_reg_in(&spi->cr1) | SPI_SPIE);
188   irq_install_handler (irq, spi_irq_handler, NULL);
189
190   return 0;
191 }
192
193
194 /*
195  * FPGA post configuration function. Blip the FPGA reset line and then see if
196  * the FPGA appears to be running.
197  */
198 int fpga_post_config_fn(int cookie)
199 {
```

```
200     int rc=0;
201
202     debug("%s:%d: FPGA post configuration\n", __FUNCTION__, __LINE__);
203
204     /* TODO: Check succes of operation */
205     debug("%s:%d: FPGA post configuration - Waiting to flush xmit queue\n",
            __FUNCTION__, __LINE__);
206   //  while(xmit_read < xmit_write);
207
208     if(xmit_read != xmit_write) {
209       printf("** WARNING - inconsistency between read and write\n");
210       rc = -1;
211     }
212
213     fpga_reg_out(&spi->cr1, fpga_reg_in(&spi->cr1) & (~SPI_SPIE));
214     irq_free_handler (irq);
215
216     return rc;
217   }
218
219   int fpga_wr8_fn(unsigned char *value, int length, int flush, int cookie)
220   {
221     int i = 0, written = length;
222     int xmit_empty = (xmit_write - xmit_read) == 0;
223
224     for(i = 0; i < length; i++)
225     {
226       if(xmit_write - xmit_read <= XMIT_BUFFER_SIZE)
227       {
228         xmit_queue[(xmit_write++) & XMIT_BUFFER_SIZE] = value[i];
229       }
230       else
231       {
232         written = i;
233         break;
234       }
235     }
236
237     if(xmit_empty)
238     {
239       xmit_read++;
240       volatile int dummy = fpga_reg_in(&spi->sr);
241       fpga_reg_out(&spi->dr, xmit_queue[(xmit_read - 1) & XMIT_BUFFER_SIZE]);
242     }
243
244     return written;
245   }
246
247
248   /*
249    * Initialize the fpga.  Return 1 on success, 0 on failure.
250    */
251   int shark_init_fpga(void)
```

```
252 {
253   debug("%s:%d: Initialize FPGA interface (relocation offset = 0x%.8lx)\n",
254         __FUNCTION__, __LINE__, gd->reloc_off);
255   fpga_init(gd->reloc_off);
256
257   fpga_serialslave_init ();
258   debug("%s:%d: Adding fpga 0\n", __FUNCTION__, __LINE__);
259   fpga_add (fpga_xilinx, &fpga[0]);
260
261   /* Configure SPI interface */
262   fpga_reg_out(&spi->brr, 0x00); // Clk = 512.6 kHz (IPB = 132 MHz), 0x74
263   fpga_reg_out(&spi->cr1, SPI_MSTR | SPI_CPOL | SPI_CPHA);
264   fpga_reg_out(&spi->cr2, SPI_SPC0);
265   fpga_reg_out(&spi->ddr, 0x0f);
266
267   fpga_reg_out(&spi->cr1, fpga_reg_in(&spi->cr1) | SPI_SPE);
268   fpga_reg_out(&spi->pdr, 0x08);
269
270   /* Configure GPIO's - Timer2 */
271   struct mpc5xxx_gpt *gpt2 = (struct mpc5xxx_gpt*) (MPC5XXX_GPT + 0x10 *
        CFG_PROGB_TIMER);
272   gpt2->emsr = 0x34; // Used as GPIO, output, log1
273
274   return 0;
275 }
276 #endif /* CONFIG_FPGA */
```

Příloha B

Programový modul Piranha

B.1 Piranha.h

```
#ifndef _PIRANHA_H_
#define _PIRANHA_H_

#include "stdio.h"

#define SLAVE_BLOCK_SIZE      0x40
#define MASTER_RAMIN_OFFSET     0x1000
#define MASTER_RAMOUT_OFFSET   0x1200

// Master status register
#define ST_DATACYCLE   0
#define ST_MODE        1
#define ST_AUTONOMIE   2
#define ST_ALLOWI      4
#define ST_ALLOWO      5

// Master mstatus register
#define MST_GO         0
#define MST_SGO        1
#define MST_RESET      3
#define MST_REQUESTI   4
#define MST_REQUESTO   5
#define MST_ERR        7

typedef struct MasterBlock
{
  char Status;
  char MStatus;
  char MCMA;
  char MCMB;
  char MCMC;
  char MCMD;
```

```
33    short int Typ;
34    char Reserved1;
35    char FifoFull;
36    char FifoEmpty;
37    char FifoSet;
38    short int MDiag;
39    short int TimeE;
40    char TimeC;
41    char Reserved2;
42    short int Timer1Hi;
43    short int Timer1Lo;
44    short int Timer2;
45    int SumMes;
46    int ErrMes;
47    short int MTimeA;
48    short int NextAdr;
49    char Change;
50    char Reserved3;
51    short int Timeout;
52    char Reserved4;
53    char CtDEREMax;
54    char NulMax;
55    char CtDotFoutMax;
56    char CtOdpFinMax;
57    char CekejDotMax;
58    short int Reserved5;
59    short int NxtTmSt01;
60    short int NxtTmSt02;
61    short int NxtTmSt03;
62    short int NxtTmSt04;
63    char Reserved6[8];
64  };
65
66  typedef struct SlaveBlock
67  {
68    char STX;
69    char CMA;
70    char CMB;
71    char CMC;
72    char CMD;
73    char ETX;
74    short int Typ;
75    short int BaseIn;
76    short int BaseOut;
77    short int Diag;
78    short int Time;
79    char TimeC;
80    char TimeDHi;
81    short int TimeDLo;
82    short int TimeL;
83    char Process;
84    char OrgProc;
85    int SumMes;
```

```
86    int ErrMes;
87    short int TimeA;
88    short NextAdr;
89    short int Reserved1;
90    short int Timeout;
91    char Addr;
92    char CtDEREMax;
93    char NulMax;
94    char InCount;
95    short int BackAdr;
96    char OutCount;
97    char ErrM;
98    short int NextTime;
99    char Reserved2;
100   char CFRAM;
101   short int FAdrFRAM;
102   char FRAM[8];
103   short int Reserved3;
104 };
105
106 class Piranha
107 {
108   public:
109     unsigned long m_BaseAddress;
110     volatile char * m_PiranhaMaster;
111     unsigned long MapProperty(const char *pathname, unsigned long psize);
112   public:
113     Piranha(char *baseAddressPath);
114
115     // Master status
116     short int GetMasterStatus();
117
118     // Methods for master configuration
119     void SetControlAndStatus(char controlAndStatus);
120     void SetMasterCMx(char MCMA, char MCMB, char MCMC, char MCMD);
121     void SetMasterCtDEREMax(char value);
122
123     // Methods for module configuration
124     void ClearModule(int position);
125     void AssignAddress(int position, char address);
126     void SetModuleCMx(int position, char CMA, char CMB, char CMC, char CMD);
127     void SetModuleETX(int position, char ETX);
128     void SetModuleType(int position, int moduleType);
129     void SetCommunicationDelays(int position, char timeC, int timeD, int timeL);
130     void SetProcess(int position, char process);
131     void SetModulePositionInCommList(int position, int nextModuleIndex, int
            prevModuleIndex);
132     void SetRAMRegions(int position, int inAddress, char inCount, int outAddress,
            char outCount);
133
134     // Methods for accessing data
135     bool LockRegion(bool outRegion);
136     bool UnlockRegion(bool outRegion);
```

```
137     void * GetModuleInput(int position);
138     void * GetModuleOutput(int position);
139 };
140 #endif //_PIRANHA_H_
```

B.2 Piranha.cpp

```
1  #include "Piranha.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7  #include <string.h>
8  #include <sys/prop.h>
9  #include <ddapi/ddapi.h>
10 #include <p4.h>
11 #include <exception>
12
13
14 /*!
15  * Creates a virtual mapping of a an area specified by property fs
16  *
17  * This method always aligns the virtual address to the physical size
18  * boundary
19  *
20  * @param *pathname - property files system path the specifies the
21  * memory mapped resource
22  * @param size - size from the property to be mapped - 0 denotes a
23  * that the value specified by property is to be used
24  * @return virtual address of the mapped memory or 0 (denotes an error)
25  */
26 unsigned long Piranha::MapProperty(const char *pathname, unsigned long psize)
27 {
28   dd_iomem_region_t mem;
29   vm_prop_ioc_t ioc;
30   int rc, fd;
31   unsigned long prop_base_addr = 0;
32
33   if ((fd = open(pathname, O_RDONLY | O_MAP)) < 0)
34   {
35     perror("Opening mmap property failed");
36     goto map_open_failed;
37   }
38
39   ioc.path = NULL;
40   /* read the property first */
41   if (ioctl(fd, PROP_READ_MEMMAP, &ioc) < 0)
```

```
42    {
43      perror("ioctl() PROP_READ_MEMMAP failed");
44      goto map_setup_failed;
45    }
46
47    /* MEMMAP begins at ioc.val.memmap.pbase and is
48     * ioc.val.memmap.poffset+ioc.val.memmap.psize bytes in size.
49     * It may be necessary to align the region size up to a
50     * multiple of page size.
51     */
52    printf("Memory mapped property: phys-base 0x%08lx, poffset 0x%08x, "
53      "size: 0x%08x\n",
54      ioc.val.memmap.pbase, ioc.val.memmap.poffset, ioc.val.memmap.psize);
55
56    /* user has not specified how many bytes from the resources
57     * should be mapped, the value from the property will be
58     * taken */
59    if (psize == 0)
60    {
61      psize = ioc.val.memmap.psize;
62    }
63
64
65    /* request a virtual address range */
66    if ((rc = dd_iomem_alloc_region(psize, psize, 0, &mem)) != 0)
67    {
68      fprintf(stderr, "dd_iomem_alloc_region failed, (%d) \"%s\"\n",
69        rc, strerror(rc));
70      goto map_setup_failed;
71    }
72
73    /* perform the mapping */
74    printf("dd_iomem_alloc_region() virt %p, size %08x\n", mem.virt, mem.size);
75    ioc.val.map_memmap.virt = (void*)mem.virt;
76    ioc.val.map_memmap.virt_size = mem.size;
77    ioc.path = 0;
78    if (ioctl(fd, PROP_MAP_MEMMAP, &ioc) < 0)
79    {
80      perror("ioctl() PROP_MAP_MEMMAP");
81      goto free_region;
82    }
83    /* Again, if poffset is not a multiple of page size,
84     * the resource begins at mem.virt+poffset and not
85     * at map_memmap.map.
86     */
87    printf("Property mapped at virt. address %p size %08x\n",
88      ioc.val.map_memmap.map, ioc.val.map_memmap.map_size);
89    prop_base_addr = (unsigned long)ioc.val.map_memmap.map;
90
91    close(fd);
92    return prop_base_addr;
93
94  free_region:
```

```
95    dd_iomem_free_region(&mem);
96  map_setup_failed:
97    close(fd);
98  map_open_failed:
99    return 0;
100 }
101
102 Piranha::Piranha(char *baseAddressPath)
103 {
104   m_BaseAddress = MapProperty(baseAddressPath, 0x2000);
105   if(m_BaseAddress == 0)
106     throw "No Piranha found";
107   m_PiranhaMaster = (volatile char*)m_BaseAddress;
108 }
109
110 short int Piranha::GetMasterStatus()
111 {
112   short int result = 0;
113
114   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
115   result = master->Status;
116   result = (result << 8) | master->MStatus;
117
118   return(result);
119 }
120
121 void Piranha::SetControlAndStatus(char controlAndStatus)
122 {
123   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
124   master->Status = controlAndStatus;
125 }
126
127 void Piranha::SetMasterCMx(char MCMA, char MCMB, char MCMC, char MCMD)
128 {
129   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
130   master->MCMA = MCMA;
131   master->MCMB = MCMB;
132   master->MCMC = MCMC;
133   master->MCMD = MCMD;
134 }
135
136 void Piranha::SetMasterCtDEREMax(char value)
137 {
138   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
139   master->CtDEREMax = value;
140 }
141
142 void Piranha::ClearModule(int position)
143 {
144   printf("Clearing module %d\n", position);
145   if( (position > 0) && (position < 63) )
146   {
```

```
147     memset((char*)m_PiranhaMaster + sizeof(SlaveBlock) * position, sizeof(
            SlaveBlock), 0);
148   }
149   else
150     throw "Out of range";
151 }
152
153 void Piranha::AssignAddress(int position, char address)
154 {
155   printf("Assign address module %d, %d\n", position, address);
156   if( (position > 0) && (position < 63) && (address > 0) && (address < 63) )
157   {
158     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
            position);
159     slave->STX = 0x40 | address;
160     slave->Addr = address;
161   }
162   else
163     throw "Out of range";
164 }
165
166 void Piranha::SetModuleCMx(int position, char CMA, char CMB, char CMC, char CMD)
167 {
168   printf("SetModuleCMx %d\n", position);
169   if( (position > 0) && (position < 63) )
170   {
171     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
            position);
172     slave->CMA = CMA;
173     slave->CMB = CMB;
174     slave->CMC = CMC;
175     slave->CMD = CMD;
176   }
177   else
178     throw "Out of range";
179 }
180
181 void Piranha::SetModuleETX(int position, char ETX)
182 {
183   printf("SetModuleETX %d\n", position);
184   if( (position > 0) && (position < 63) )
185   {
186     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
            position);
187     slave->ETX = ETX;
188   }
189   else
190     throw "Out of range";
191 }
192
193 void Piranha::SetModuleType(int position, int moduleType)
194 {
195   printf("SetModuleType %d\n", position);
```

```
196   if( (position > 0) && (position < 63) )
197   {
198     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
          position);
199     slave->Typ = moduleType;
200   }
201   else
202     throw "Out of range";
203 }
204
205 void Piranha::SetCommunicationDelays(int position, char timeC, int timeD, int
      timeL)
206 {
207   if( (position > 0) && (position < 63) )
208   {
209     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
          position);
210     slave->TimeC = timeC;
211     slave->TimeDHi = timeD > 16;
212     slave->TimeDLo = timeD & 0xffff;
213     slave->TimeL = timeL;
214   }
215   else
216     throw "Out of range";
217 }
218
219 void Piranha::SetProcess(int position, char process)
220 {
221   if( (position > 0) && (position < 63) )
222   {
223     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
          position);
224     slave->Process = process;
225   }
226   else
227     throw "Out of range";
228 }
229
230 void Piranha::SetModulePositionInCommList(int position, int nextModuleIndex, int
      prevModuleIndex)
231 {
232   if( (position > 0) && (position < 63) )
233   {
234     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
          position);
235     slave->NextAdr = nextModuleIndex;
236     slave->BackAdr = prevModuleIndex;
237   }
238   else
239     throw "Out of range";
240 }
241
```

```
242 void Piranha::SetRAMRegions(int position, int inAddress, char inCount, int
        outAddress, char outCount)
243 {
244   if( (position > 0) && (position < 63) )
245   {
246     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
            position);
247     slave->BaseIn = inAddress;
248     slave->InCount = inCount;
249     slave->BaseOut = outAddress;
250     slave->OutCount = outCount;
251   }
252   else
253     throw "Out of range";
254 }
255
256 bool Piranha::LockRegion(bool outRegion)
257 {
258   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
259
260   master->Status = master->Status & ((outRegion) ? ~(1 << ST_ALLOWO) : ~(1 <<
          ST_ALLOWI));
261   int mask = outRegion ? (1 << MST_REQUESTO) : (1 << MST_REQUESTI);
262
263   int iCounter = 100;
264   while( (iCounter-- > 0) && (master->MStatus & mask) );
265
266   usleep(10);
267
268   return( (iCounter > 0) && ((master->MStatus & mask) == 0) );
269 }
270
271 bool Piranha::UnlockRegion(bool outRegion)
272 {
273   MasterBlock *master = (MasterBlock*)m_PiranhaMaster;
274
275   master->Status = master->Status | ((outRegion) ? (1 << ST_ALLOWO) : (1 <<
          ST_ALLOWI));
276   return(true);
277 }
278
279 void * Piranha::GetModuleInput(int position)
280 {
281   void *result = NULL;
282   if( (position > 0) && (position < 63) )
283   {
284     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
            position);
285     result = (void*)(m_PiranhaMaster + MASTER_RAMIN_OFFSET + slave->BaseIn);
286   }
287   else
288     throw "Out of range";
289   return(result);
```

```
290 }
291
292 void * Piranha::GetModuleOutput(int position)
293 {
294   void *result = NULL;
295   if( (position > 0) && (position < 63) )
296   {
297     SlaveBlock *slave = (SlaveBlock*)(m_PiranhaMaster + sizeof(SlaveBlock) *
          position);
298     result = (void*)(m_PiranhaMaster + MASTER_RAMOUT_OFFSET + slave->BaseOut);
299   }
300   else
301     throw "Out of range";
302   return(result);
303 }
```

B.3 main.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #ifdef POSIXDEBUG
7  #include <sys/debug.h>
8  #endif
9  #include <sys/prop.h>
10 #include <ddapi/ddapi.h>
11
12 #include <p4.h>
13
14 #include "Piranha.h"
15
16 #define PIRANHA_PROPFS_PATH "/prop/app/posix/posix/piranha/mem"
17
18 int
19 main(void)
20 {
21   int stop = 0;
22
23   puts("Piranha demo application starting up.");
24 #ifdef POSIXDEBUG
25   gdb_breakpoint();
26 #endif
27
28   Piranha *Master = NULL;
29
30   try
```

```
31    {
32      Master = new Piranha(PIRANHA_PROPFS_PATH);
33
34      if(Master->GetMasterStatus() & (1 << MST_RESET))
35      {
36        Master->ClearModule(1);
37        Master->ClearModule(2);
38
39        Master->AssignAddress(1, 1);
40        Master->SetModuleCMx(1, 0, 0, 0, 0);
41        Master->SetModuleETX(1, 0x02);
42        Master->SetModuleType(1, 0x8002);
43        Master->SetCommunicationDelays(1, 0, 0x1ff, 0);
44        Master->SetProcess(1, 0x5);
45        Master->SetModulePositionInCommList(1, 1, 1);
46        Master->SetRAMRegions(1, 0, 0, 0, 1);
47
48        Master->AssignAddress(2, 2);
49        Master->SetModuleCMx(2, 0, 0, 0, 0);
50        Master->SetModuleType(2, 0x4002);
51        Master->SetCommunicationDelays(2, 0, 0x1ff, 0);
52        Master->SetProcess(2, 0x5);
53        Master->SetModulePositionInCommList(2, 0, 0);
54        Master->SetRAMRegions(2, 0, 1, 0, 0);
55
56        Master->SetMasterCMx(0, 1, 0, 0);
57        Master->SetMasterCtDEREMax(0);
58        Master->SetControlAndStatus(0x32);
59        usleep(100000);
60        Master->SetControlAndStatus(0x31);
61
62        char *input = (char*)Master->GetModuleInput(2);
63        char *output = (char*)Master->GetModuleOutput(1);
64
65        printf("Input: %x, Output: %x\n", input, output);
66
67        *output = 0x3c;
68
69        int iCounter = 1;
70        while (!stop) {
71          iCounter--;
72
73          if(Master->LockRegion(false))
74          {
75            char Inputs = *input;
76            Master->UnlockRegion(false);
77
78            if(iCounter == 0)
79            {
80              printf("Master status: 0x%x, Input (D7..D0): ", Master->
                    GetMasterStatus());
81              for(int i = 7; i >= 0; i--)
82                printf( (Inputs & (1 << i)) ? "1" : "0");
```

```
83              printf("\n");
84              iCounter = 1000000;
85            }
86
87            if(Master->LockRegion(true))
88            {
89              *output = Inputs;
90              Master->UnlockRegion(true);
91            }
92          }
93        }
94      }
95      else
96        throw "Master unavailable";
97    }
98    catch(const char *str)
99    {
100     printf("%s\n", str);
101   }
102
103   if(Master != NULL)
104     delete(Master);
105   return 0;
106 }
```

Příloha C

Obsah přiloženého CD

K této práci je přiloženo CD, na kterém jsou uloženy zdrojové kódy.

- *dp_2010_michal_hrouda.pdf* - Tato práce ve formátu PDF
- datasheets
  Katalogové listy použité při psaní této práce
  - mpc5200 - Katalogové listy a aplikační poznámky k procesoru MPC5200
  - spartan3a - Katalogové listy a aplikační poznámky k FPGA Spartan-3A
- pikeos - Zdrojový kód realizované aplikace pro PikeOS
- u-boot - Patch pro oficiální release U-Boot verze 1.3.2 doplňující podporu pro desku Shark a konfiguraci FPGA