VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

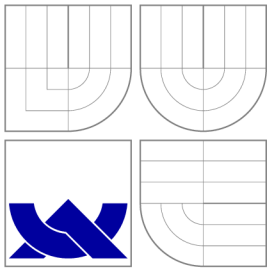
KOMUNIKACE S FPGA OBVODY POMOCÍ DMA PL330

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

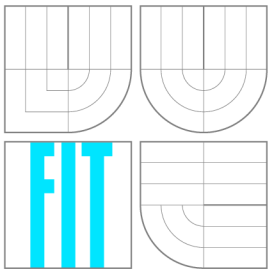
AUTOR PRÁCE
AUTHOR

JAN HAVRAN

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOMUNIKACE S FPGA OBVODY POMOCÍ DMA PL330

COMMUNICATION WITH FPGA CIRCUITS USING DMA PL330

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HAVRAN
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN VIKTORIN
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá návrhem a implementací komunikace s FPGA pomocí obvodu PL330. Pro zvolené řešení byly implementovány dva jaderné moduly v jazyce C pro OS Linux. Tyto moduly poskytují rozhraní pro efektivní komunikaci mezi user space aplikací a jednotkou FPGA.

Abstract

This thesis describes the design and implementation of communication with FPGA using PL330 circuit. For this solution were implemented two kernel modules for OS Linux in language C. These modules provides interface for effective communication between user space application and FPGA.

Klíčová slova

Linux, jádro, ovladač, kernel space, user space, PL330, DMA, FPGA, znakové zařízení, DMA Engine, AXI sběrnice, peripheral request interface.

Keywords

Linux, kernel, driver, kernel space, user space, PL330, DMA, FPGA, character device, DMA Engine, AXI bus, peripheral request interface.

Citace

Jan Havran: Komunikace s FPGA obvody pomocí DMA PL330, bakalářská práce, Brno, FIT VUT v Brně, 2015

Komunikace s FPGA obvody pomocí DMA PL330

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Viktorina. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Havran
18. května 2015

Poděkování

Chtěl bych poděkovat Ing. Janu Viktorinovi za trpělivost a cenné rady, které mi byly poskytnuty při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Díky rychlému vývoji výpočetní techniky posledních několika (desítek) let je možné realizovat projekty, které by se před nedávnou dobou zdály nepředstavitelné. Nejen, že se stále daří zvyšovat výkon procesorů, zároveň ale také dochází ke zmenšování plochy čipů a snižování spotřeby. Tyto skutečnosti umožnily výrobcům výpočetní techniky instalovat mikropočítače do téměř každého zařízení. Dnes jsou výkonné procesory nejen v mobilních telefonech, ale i v televizorech, tzv. chytrých hodinkách a několik se jich nachází v automobilech.

Trendem dnešní doby jsou integrované obvody, kde se na jediném čipu nachází procesorová jednotka, paměti a různé typy periférií – jedná se o tzv. System on Chip (SoC). Takové řešení se používá zejména v situacích, kde jsou parametry jako cena a spotřeba hlavními kritérii. Využití je zejména ve vojenském nebo herním průmyslu (konzole), nebo v komunikačních či síťových zařízení.

Pro specifické úlohy, kde je kritický čas výpočtu a spotřeba, je součástí čipu také obvod FPGA[1]. Obvod FPGA je užíván pro svoji flexibilitu a zároveň výhodu v podobě rychlosti hardwarovách výpočtů. Dále existují komponenty, které obsahují SoC spolu s integrovaným FPGA. Tyto komponenty tak najdou uplatnění například ve zpracování velkého objemu dat jako audio a video či síťového provozu. Integrované FPGA spolu s procesorem lze nalézt například na platformách Xilinx Zynq a Altera SoC FPGA. Pro získání dat z FPGA pro analýzu či případné další zpracování, nebo naopak k zaslání dat do FPGA, je možné využít různých typů přístupu. Kromě přímého přístupu z procesou (softwarové řešení), které má nevýhodu v zatěžování procesoru, se nabízí možnost využití obvodu PL330, který je integrovaný na obou zmiňovaných platformách. Tento programovatelný DMA obvod je již součástí čipu a přístupný i z programovatelného pole. Při využití DMA se tak může procesor v čase přenosu věnovat jiným úlohám.

[1] Field-programmable gate array

Kapitola 2

Analýza

Tato kapitola se zabývá analýzou platformy Xilinx Zynq a vývojových prostředků operačního systému Linux. Je důležité zjistit, jak probíhá komunikace mezi zařízeními, a to jak na úrovni hardware, tak software, jakých technologích se využívá a identifikovat případné problémy, které by mohly mít negativní dopad na řešení.

2.1 Hardware

V této podkapitole budou popsány některé důležité pojmy, jejichž znalost bude vyžadována v dalších kapitolách. Jelikož se jedná o stěžejní problematiku pro tuto bakalářskou práci, bude jim zde věnován krátký úvod do problematiky.

2.1.1 Procesory ARM

Procesory ARM[1], které vyvíjí firma ARM Holdings, se řadí mezi procesorovou architekturu RISC[2] (jak napovídá i název). Ačkoliv ARM Holdings dříve procesory ARM přímo vyráběla, tak později přešla na model licencování. To znamená, že výrobcům hardware jsou pronajímány licence na tuto architekturu, což na jedné straně výrobcům hardware ušetří finance za vývoj vlastní platformy (a instrukční sady), a na straně druhé umožní optimalizovat procesory přímo na požadovaný hardware. I proto jsou ARM procesory hojně využívány v mobilních zařízeních a vestavěných systémech, kde mohou efektivně plnit svoji roli s výhodou nízké spotřeby.

RISC architektura je specifická svojí redukovanou instrukční sadou, která je hardwarově implementovaná pevně daným automatem (narozdíl od složité architektury CISC, kde jsou mnohdy složité instrukce složeny z několika mikroinstrukcí). Z toho vyplývají výhody v podobě snadnější implementace kompilátorů, menší velikosti výsledného čipu a nižší spotřeby. Dále je RISC architektura známá díky obecně většímu počtu univerzálních registrů (oproti architektuře CISC). Další odlišností je například to, že pro práci s pamětí existuje v RISC architekturách minimální množství instrukcí: instrukce *Load* pro čtení z paměti, a instrukce *Store* pro zápis do paměti. Kromě instrukcí podmíněných skoků, které jsou součástí naprosté většiny architektur, přinášejí procesory ARM také podmíněné instrukce. Ty mohou zefektivnit kód oproti jiným architekturám, kde by byly potřeba podmíněné skoky, různá porovnávání atd.

[1] Advanced RISC Machine
[2] Reduced Instruction Set Computing

2.1.2 Direct Memory Access

Pro řízení přenosu dat mezi periférierni a pamětí se využívá různých principů. Data lze přenášet s využitím procesoru, a to buď pomocí speciálních instrukcí *IN* a *OUT*, které přenášejí data z isolovaných vstupů, respektive do isolovanaých výstupů. Tyto vstupy a výstupy mají vlastní adresový prostor (v takovém případě jsou tedy dva adresové prostory: prostor pro paměť a pro isolované vstupy a výstupy). Pokud jsou registry periferního zařízení mapovány do paměťového adresového prostoru, lze využít instrukcí pro práci s pamětí (lze tak využít většího množství instrukcí pro operace čtení a zápisu). Obě dvě řešení ale vytěžují procesor, který musí zbytečně čekat na dokončení pomalé paměťové operace. Místo přenosu dat řízeným procesorem lze využít principu DMA[3]. DMA provádí přenos z registrů periférních zařízení do paměti a zpět bez zatížení procesoru. Procesor je pouze o takovém přenosu informován (a v závislosti na typu sběrnice se od sběrnice odpojí) – může tedy vykonávat jiné instrukce v době přenosu dat[4].

V případě principu DMA můžeme rozlišovat více přístupů. Prvním typem je tzv. bus mastering, kdy mají klienti sběrnice schopnost řídit přenosy přes sběrnici (tuto schopnost má obvykle pouze procesor). Periférie, která chce komunikovat, si vyžádá řízení sběrnice a poté jako master řídí autonomně přenosy. K takto řízeným přenosům musí být vybavena sběrnice (například PCI) patřičnými signály. Pokud může být více zařízení v roli master, tak je v takovém případě nutné řešit arbitraci na sběrnici. Druhým typem je přenos Central DMA. V tomto případě se na sběrnici nachází další řadič – řadič DMA (takto je vybavena například sběrnice ISA). DMAC[4] je jednotka na sběrnici, která má schopnost autonomního řízení přenosů (je v roli master). Ostatní zařízení na sběrnici jsou v roli slave. Zařízení si vždy musí zažádat jednotku DMAC o přenos.[4].

2.1.3 FPGA

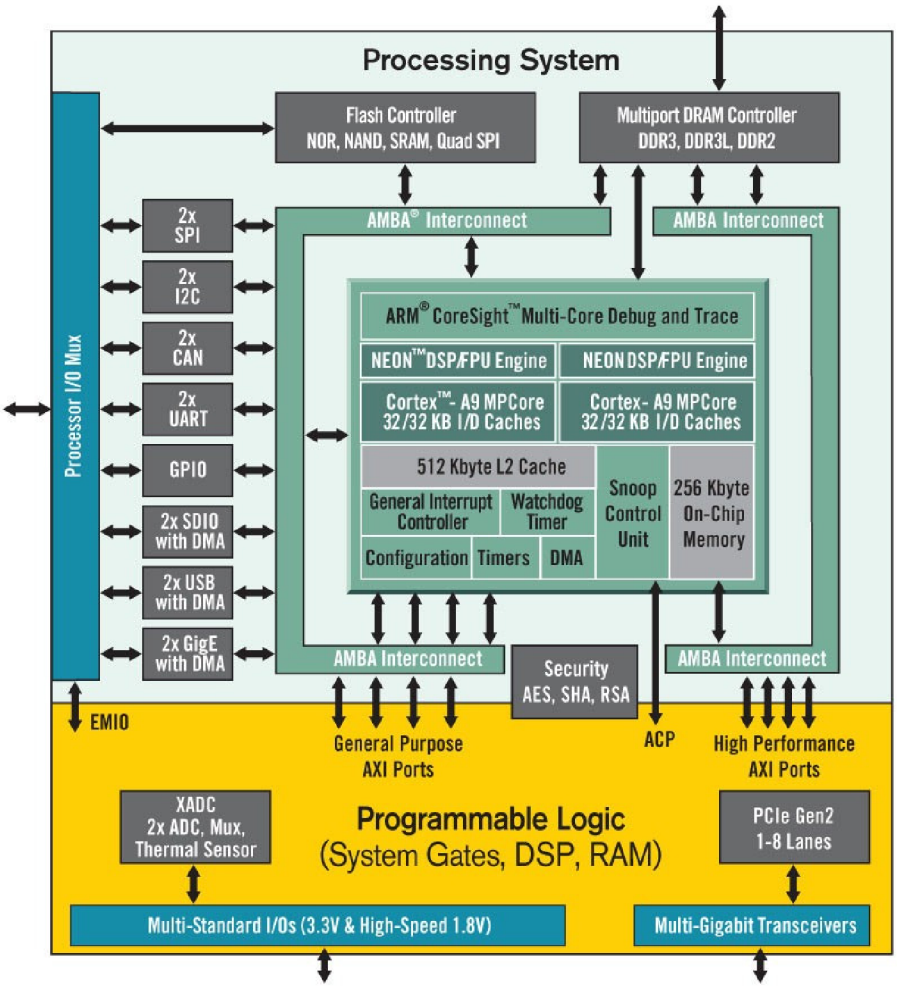
FPGA, neboli programovatelné hradlové pole, je integrovaný obvod obsahující programovatelné (logické) bloky, díky čemuž je možné obvod libovolně konfigurovat i po opuštění výrobní továrny. Tyto bloky jsou umístěny do konfigurovatelné matice propojů, díky čemuž je možné vytvářet spoje mezi bloky, naprogramovat obvod pro řešení konkrétního algoritmu a využít tak rychlosti a paralelizaci hardwarové implementace. Mezi nejznámější jazyky pro konfiguraci FPGA patří VHDL[5] a Verilog. FPGA jednotka bývá kombinována s procesory (nejčastěji ARM) a jinými periférierni v jediný čip (SoC). Mezi přední výrobce těchto hybridních architektur patří společnosti Xilinx a Altera.

2.2 Platforma Xilinx Zynq

Xilinx je americká firma zaměřená na vývoj logických obvodů a patří mezi nejvýznamnější výrobce programovatelných hradlových polí (FPGA). Platforma Xilinx Zynq patří mezi architektury AP SoC[6] a je zaměřena na výkonné vestavěné systémy. Jedná se o procesor-centrickou architekturu[9] (firma vyvíjí i FPGA-centrické). Platforma obsahuje dvoujádrový ARM procesor Cortex-A9 MPCore, programovatelnou logiku (PL), jako je například FPGA, cache PL310, paměť PL353 nebo DMA jednotka PL330. Dále je obsažen USB 2.0 řadič,

[3]Direct Memory Access - přímý přístup do paměti
[4]DMA Controller
[5]VHSIC Hardware Description Language
[6]All Programmable SoC

watchdog, gigabit ethernet, GPIO[7] a další. Blokový diagram architektury je zobrazen na obrázku 2.1 (převzatý z webových stránek Xilinxu[8]).



Obrázek 2.1: Blokový diagram Xilinx Zynq 7000

2.2.1 AXI sběrnice

Součástí Xilinx Zynq je specifikace AMBA[9], jejíž součástí jsou sběrnice APB[10], AHB[11] a AXI[12]. AXI sběrnice je zaměřena na nízkou odezvu, vysokou frekvenci, ale zároveň dodržuje určitou zpětnou kompatibilitu s rozhraními APB a AHB. AXI má samostatné kanály pro adresovou a datovou část a také části pro čtení, zápis a odpověď. Jedná se o následující kanály:

- **Read Address Channel** – kanál pro nastavení adresy pro čtení dat.

[7] General-purpose input/output
[8] http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000
[9] Advanced Microcontroller Bus Architecture
[10] Advanced Peripheral Bus
[11] Advanced High-performance Bus
[12] Advanced eXtensible Interface

- **Write Address Channel** – kanál pro nastavení adresy pro zápis dat.
- **Read Data Channel** – kanál pro čtení dat.
- **Write Data Channel** – kanál pro zápis dat.
- **Write Response Channel** – kanál pro odpovědi.

AXI podporuje přenos nezarovnaných dat a burst přenosy, což je obecně princip zaslání většího počtu dat bez opakovaného zahajování přenosů. Burst tedy snižuje celkovou režii přenosu. Při komunikaci AXI využívá principů *handshaking*, což umožňuje zpomalovat rychlost sběrnice. AXI *handshaking* pracuje tak, že jakmile jsou data připravena na zdroji přenosu, je nastaven signál *VALID*. Cíl přenosu na tento signál reaguje a pokud je připraven, nastaví signál *READY* do logické jedničky. Zahájení přenosu proběhne nastavením obou signálů na úroveň logické jedničky.

AXI přenáší data pomocí burst přenosů. V případě AXI je pouze na počátku každého burstu přenesena zařízením master zdrojová adresa prvního přenášehého bajtu (odvození dalších adres je pak záležitostí zařízení slave). Pro burst přenos je využíván jeden kanál na data a samostatný kanál na signalizaci dokončení přenosu. Délku burst přenosu lze nastavit na hodnoty v rozmezí 1 až 16 (platí pro AXI3), přičemž velikost jednoho přenosu je 8 až 1024 bitů[1]. Lze také měnit konfiguraci mezi inkrementálním a neinkrementálním burstem (burst s pevnou adresou).

2.3 Obvod PL330

PL330 je jednotka DMAC firmy ARM. Jednotka je kompatibilní se specifikací AMBA, ze které využívá AXI sběrnici pro přenosy DMA a dvě sběrnice APB pro řízení přenosu. Tento obvod dále podporuje ARM technologii TrustZone (aplikace může přepínat mezi dvěma bezpečnostními stavy podle potřeby), kdy jedna sběrnice APB je využita pro zabezpečený režim a druhá pro nezabezpečený[2]. PL330 podporuje čtyři typy přenosu:

- **paměť – paměť** – slouží pro přenos dat mezi dvěmi adresami paměti.
- **paměť – zařízení** – přenáší data z adresy paměti do registrů zařízení.
- **zařízení – paměť** – přenáší data z registrů zařízení na adresu paměti.
- **scatter-gather** – jedná se o speciální typ přenosu. Používá se v případě, kdy se různé části dat nachází na různých místech. Místo opakovaného přenášení jednotlivých částí dat je možné využít tohoto principu, kdy se místo jedné adresy použije vektor adres. Díky tomuto přístupu lze přenos nakonfigurovat pouze jednou pro různé adresy datových oblastí. Podle rozsypání/sesbírání dat se princip nazývá *scatter/gather*.

K produktům firmy ARM jsou dodávány podrobné materiály, umožňující (v rámci licenční politiky) si obvod různě upravit. Při vlastním návrhu obvodu PL330 je možné jednotku různě nakonfigurovat. PL330 podporuje modifikaci následujících parametrů (nejedná se o celý výčet):

- Počet rozhraní Peripheral request interface
- Velikost fronty instrukcí pro čtení/zápis

- Počet řádků instrukční cache

Konfiguraci pro konkrétní jednotku PL330 lze získat z konfiguračních registrů (Configuration Registers), viz sekce 2.3.3.

PL330 má k dispozici až 8 nezávislých DMA kanálů pro přenos dat. Každý z těchto osmi kanálů je řízený samostatným vláknem, které vykonává jednotlivé instrukce (jednotka PL330 má vlastní instrukční sadu, viz sekce 2.3.2). Kromě vláken kanálů disponuje jednotka ještě manager vláknem, které nemá přiděleno žádný DMA kanál, ale které slouží k řízení ostatních vláken (lze tak ostatní vlákna spustit, případně zastavit).

2.3.1 Dostupná rozhraní

Obvod PL330 je připojen k několika sběrnicím. Jedná se o:

- AXI
- Non-secure APB
- Secure APB
- Peripheral request interaface
- Interrupt

AXI master interface, přes kterou je DMA jednotka připojena k AXI Interconnect, byla popsána v sekci 2.2.1.

Dalšími rozhraními jsou APB (zabezpečnené a nezabezpečené). Pomocí APB je možné přistupovat k registrům DMA obvodu (registry popsány v sekci 2.3.3). Pro každé APB rozhraní má jednotka PL330 alokované 4KiB adresového prostoru - jednotlivé registry jsou tak adresovatelné v rozsahu 0x000 až 0xFFF. Přes APB rozhraní lze přímo nahrát do DMA jednotky tři různé instrukce: `DMAGO`, `DMASEV` a `DMAKILL`.

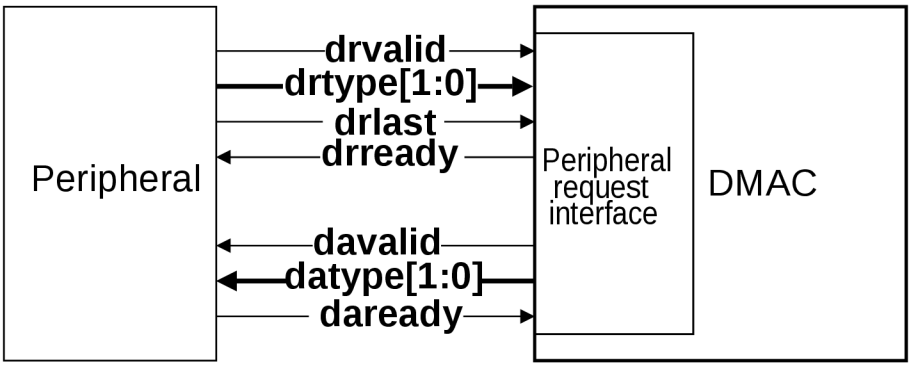
Interrupt interface umožňuje jednotlivým vláknům jednotky vyvolat přerušení (pokud je nastaven příslušný bit v registru Interrupt Enable) a informovat tak procesor například o dokončení přenosu.

Peripheral request interface (PRI) rozšiřuje komunikační kanál o několik dalších signálů, které jsou znázorněny na obrázku 2.2. Signály s prefixem `dr` slouží periférii k zaslání požadavků. Signály s prefixem `da` naopak slouží k potvrzování jednotkou DMAC. Signály `valid` a `ready` jsou využívány pro *handshaking* (obdobně jako AXI sběrnice). Signál `drtype` (2 bitový) slouží k zaslání požadavku na přenos typu burst, případně single (po dokončení takového přenosu odpoví jednotka PL330 signálem `datype` stejného typu), případně jako potvrzení flush požadavku zaslaného jednotkou PL330 skrze signál `datype`. Dále je ještě přítomen signál `drlast` (pouze v jednom směru) umožňující signalizovat konec přenosu.

S některými signály umí přímo pracovat některé PL330 instrukce. Zároveň nemusí vždy platit, že každý DMA kanál má PRI (například v konfiguraci pro Altera SoC FPGA disponují PRI rozhraním pouze první čtyři kanály).

2.3.2 Instruční sada

Jak bylo popsáno v sekci 2.3, PL330 disponuje několika vlákny, které řídí jednotlivé DMA kanály, a jedním řídícím vláknem. Každé vlákno obsahuje vlastní registr PC (program counter). DMA kontrolér poskytuje vlastní instrukční sadu pro řízení přenosu (viz tabulka



Obrázek 2.2: Peripheral Request Interface

2.1). Instrukce mají proměnnou délku 1 až 6 bajtů, přičemž 8 bitů tvoří operační kód instrukce. Pomocí instrukcí je možné DMAC nakonfigurovat například pro určitou délku AXI burstů, šířku jednoho přenosu, nastavení přerušení atd. PL330 udržuje vnitřní signál `request_flag`, který je používán u podmíněných instrukcí. Tento signál může nastavit pouze instrukce `DMAWFP` (viz dále).

Název	vlákno	Název	vlákno
`DMAADDH`	C	`DMAMOV`	C
`DMAEND`	M/C	`DMANOP`	M/C
`DMAFLUSHP`	C	`DMARMB`	C
`DMAGO`	M	`DMASEV`	M/C
`DMALD`	C	`DMAST`	C
`DMALDP`	C	`DMASTP`	C
`DMALP`	C	`DMASTZ`	C
`DMALPEND`	C	`DMAWFE`	M/C
`DMALPFE`	C	`DMAWFP`	C
`DMAKILL`	M/C	`DMAWMB`	C

Tabulka 2.1: Instrukční sada obvodu PL330 – M je řidící vlákno, C je vlákno kanálu

Následuje stručný popis instrukcí z tabulky 2.1:

- `DMAADDH <registr>, <hodnota>` – přičte 16 bitovou *hodnotu* k horní polovině 32 bitového *registru*.
- `DMAEND` – ukončení činnosti DMA vlákna.
- `DMAFLUSHP <periférie>` – zaslání signálu `daflush=flush request` *periférii* a čekání na odpověď v podobě signálu `drtype=flush acknowledge`.
- `DMAGO <kanál>, <hodnota> [,ns]` – spustí program *kanálu* na adrese *hodnota*. V závislosti na parametru *ns* bude program operovat ve stejném režimu jako manager vlákno (pokud parametr není zadán, bude operovat v nezabezpečeném režimu).
- `DMALD[S|B]` – načtení 32 bitové hodnoty. Volitelný parametr určí podmíněné provedení instrukce (pokud parametr neodpovídá signálu `request_flag`, je provedena instrukce `DMANOP`).

- `DMALDP<S|B> <periférie>` – načtení 32 bitové hodnoty. Povinný parametr určí podmíněné provedení instrukce (pokud parametr neodpovídá signálu `request_flag`, je provedena instrukce `DMANOP`). Po provedení instrukce je informována *periférie* signálem `datype=single/burst` (typ signálu odpovídá parametru `S` (single) nebo `B` (burst)).
- `DMALP <počet>` – Zahájení smyčky o *počtu* iterací.
- `DMALPEND[S|B]` – Ukončení smyčky. Pokud byla smyčka zahájena instrukcí `DMALP`, je smyčka ukončena po dosažení daného počtu iterací. V případě zahájení smyčky instrukcí `DMALPFE` (popsaná dále), je smyčka ukončena signálem `drlast`. Volitelný parametr určí podmíněné provedení instrukce (pokud parametr neodpovídá signálu `request_flag`, je provedena instrukce `DMANOP`).
- `DMALPFE` – nejedná se o skutečnou instrukci. Slouží pouze pro informaci kompilátoru o tom, jak byla smyčka započata (důležité pro instrukci `DMALPEND`).
- `DMAKILL` – ukončení vlákna, které provedlo tuto instrukci.
- `DMAMOV <registr>, <hodnota>` – uložení *hodnoty* do daného *registru* (registr může být pouze Source Address Register, Destination Address Register nebo Channel Control Register – registry jsou popsány v sekci 2.3.3).
- `DMANOP` – žádná operace.
- `DMARMB` – čtecí bariéra. DMA vlákno čeká dokud nejsou provedeny všechny AXI operace čtení daného kanálu.
- `DMASEV <hodnota>` – signalizace události číslo *hodnota*.
- `DMAST[S|B]` – uložení 32 bitové hodnoty. Volitelný parametr určí podmíněné provedení instrukce (pokud parametr neodpovídá signálu `request_flag`, je provedena instrukce `DMANOP`).
- `DMASTP<S|B> <periférie>` – uložení 32 bitové hodnoty. Povinný parametr určí podmíněné provedení instrukce (pokud parametr neodpovídá signálu `request_flag`, je provedena instrukce `DMANOP`). Po provedení instrukce je informována *periférie* signálem `datype=single/burst` (typ signálu odpovídá parametru `S` (single) nebo `B` (burst)).
- `DMASTZ` – uložení hodnoty `0`.
- `DMAWFE <hodnota>` – čekání na signalizaci události s číslem *hodnota*. Více vláken může čekat na stejnou událost.
- `DMAWFP<S|B|P> <periférie>` – čekání na zaslání požadavku typu burst, případně single dané *periférie*. Tato instrukce jako jediná nastavuje `request_flag`. Parametr `S` určuje čekání na single nebo burst request a nastavení `request_flag` na single, parametr `B` čeká na burst request a nastavuje `request_flag` na typ burst a parametr `P` čeká na single nebo burst request a podle typu requestu nastaví `request_flag` na single nebo burst.
- `DMAWMB` – zapisovací bariéra. DMA vlákno čeká dokud nejsou provedeny všechny AXI operace zápisu daného kanálu.

2.3.3 Registry

DMAC má k dispozici 4 KiB adresovatelné paměti od adresy 0x000 až do 0xFFF. Kontrolér obsahuje hned několik 32 bitových registrů. Registry jsou adresovány od počátku adresového prostoru (0x000), přičemž adresa dalšího registru má offset zvětšený o 4 bajty (32 bitů). Všechny registry jsou přístupné jak z CPU, tak z jednotky PL330 (některé instrukce PL330 pracují přímo s registry). DMAC jednotka obsahuje tyto základní registry[2]:

- **DS** (DMA Status Registr) – poskytuje informace o konfiguraci a stavu kontroléru.
- **DPC** (DMA Program Counter Register) – programový čítač.
- **INTEN** (Interrupt Enable Register) – bitové povolení jednotlivých přerušení.
- **ES** (Event Status Register) – poskytuje stav požadavků událostí/přerušení.
- **INTSTATUS** (Interrupt Status Register) – stav aktivního přerušení.
- **INTCLR** (Interrupt Clear Register) – řízení vymazání přerušení.
- **FSM** (Fault Status DMA Manager Register) – chybový stav řídicího vlákna (je využit jediný bit).
- **FCM** (Fault Status DMA Channel Register) – chybový stav jednotlivých kanálů (pro každý kanál je vyhrazen jeden bit).
- **FTM** (Fault Type DMA Manager Register) – značí typ chyby, která nastala v řídícím vláknu.

Kromě těchto registrů DMAC obsahuje ještě několik dalších, které jsou jednotlivě dostupné pro každý z osmi kanálů (každý registr má tedy svoji variantu ve všech kanálech 0 až 7). V následujícím výčtu registrů písmeno n značí číslo daného kanálu:

- **FTCn** (Fault Type DMA Channel Registers) – signalizuje typ chyby, která nastala na daném kanálu.
- **CSn** (Channel Status Registers) – stav programu na DMA kanálu.
- **CPCn** (Channel Program Counter Registers) – programový čítač daného kanálu.
- **SA_n** (Source Address Registers) – adresa zdrojových dat pro DMA kanál.
- **DA_n** (Destination Address Registers) – adresa cílových dat pro DMA kanál..
- **CC_n** (Channel Control Registers) – slouží pro řízení přenosu dat. Má část pro řízení zdrojových a cílových dat. Lze nastavit například velikost burst přenosu.
- **LC0_n** (Loop Counter 0 Registers) – poskytuje aktuální stav prvního čítače cyklů.
- **LC1_n** (Loop Counter 1 Registers) – poskytuje aktuální stav druhého čítače cyklů.

Všechny výše zmíněné registry jednotlivých kanálů jsou přes APB rozhraní přístupné pouze pro čtení. Hodnoty těchto registrů lze měnit buď jednotlivými instrukcemi (`DMAWFP` mění registry SA, DA a CC, instrukce `DMALP` nastavuje registry LC0 a LC1) nebo jsou nastavovány přímo obvodem PL330. Z registru CS lze, mimo jiné, vyčíst například aktuální status kanálu, nebo na jakou událost kanál čeká, pokud je ve stavu Waiting. Registr CC je rozdělen na zdrojovou část (source) a cílovou část (destination). Konfigurací tohoto registru lze nastavit:

- **Burst len** - počet datových přenosů během jednoho burstu. AXI sběrnice (a také obvod PL330) dovoluje hodnotu 1 až 16, přičemž hodnota 1 značí single přenos.
- **Burst size** - velikost jednotlivých přenosů. Lze nastavit 1 až 128 bajtů. Celkový počet přenesených bajtů během jediného burstu je tedy dán součinem velikostí a délky burstu (během jednoho burstu je tak možné přenést až 2KiB dat).
- **Inc** - nastaví režim přenosu (inkrementační/fixní adresa). Pro inkrementanční adresu instrukce `DMALD` a `DMAST` automaticky zvýší adresu v registru SA/DA o velikost přenosu.
- **Cache** - konfigurace cache.
- **Prot** - konfigurace ARPROT[2:0]/AWPROT[2:0] signálů. AxProt[0] určuje neprivilegovaný/privilegovaný přístup, AxProt[1] zabezpečený/nezabezpečený (secure/non-secure) přístup a AxProt[2] přístup dat/instrukcí.

Dále PL330 disponuje čtyřmi Debug registry:

- **DBGSTATUS** (Debug Status Register) – má definovaný pouze nultý bit, který určuje debug stav DMAC (nečinný/aktivní).
- **DBGCMD** (Debug Command Register) – definovány pouze první dva bity. Zápisem nulové hodnoty do tohoto registru se spustí DMA instrukce obsažená v registrech `DBGINST0` a `DBGINST1`.
- **DBGINST0** (Debug Instruction-0 Register) – obsahuje první 2 bajty debug instrukce, dále určuje typ spouštěného vlákna (manager nebo kanál) a číslo kanálu, v případě, že je typ vlákna kanál.
- **DBGINST1** (Debug Instruction-1 Register) – obsahuje zbývající 4 bajty debug instrukce.

Nakonec PL330 obsahuje konfigurační registry (Configuration Register 0 až 4 a Configuration Register Dn), obsahující aktuální konfiguraci danou výrobcem, čtyři 8 bitové registry Peripheral Identification Registers 0-3 (lze je číst jako jeden 32 bitový registr) a čtyři 8 bitové registry PrimeCell Identification Registers 0-3 (opět je lze číst jako jeden 32 bitový registr).

2.3.4 Programování jednotky PL330

Program může být do PL330 nahrán dvěma způsoby:

1. **Bootování z určité adresy** – po resetu DMAC jednotky se začne provádět program, který je uložen na adrese danou registrem Configuration Register 2.
2. **Ručním nahráním příslušných instrukcí** – jednotlivé instrukce lze nahrát do debug registru a poté spustit.

Mezi těmito dvěma způsoby nelze libovolně přepínat, neboť daný způsob je zakódován přímo do hardwaru. Xilinx Zynq i Altera SoC FPGA používají druhý způsob, tedy ruční náhrání instrukcí, proto zde bude popsána tato varianta.

Jak již bylo řečeno v sekci 2.3.1, jediné instrukce, které lze přes APB rozhraní nahrát do Debug registrů, jsou `DMAGO`, `DMASEV` a `DMAKILL`. Právě instrukce `DMAGO` ale umožňuje spouštění libovolného programu, který může být předem sestaven podle požadavků.

Na adresu, která je přístupná pro jednotku PL330, je třeba uložit DMA program. Jedná se v podstatě o řetězec bajtů reprezentujících jednotlivé instrukce. Jakmile je program sestaven, je třeba nahrát do Debug registrů 0 a 1 instrukci `DMAGO`. Tato instrukce obsahuje ID kanálu, který bude spuštěn, a 32 bitovou adresu programu. Je nezbytné, aby poslední instrukcí programu byla instrukce `DMAEND`, jinak by mohlo nastat nepředvídatelné chování programu. Dále je vhodné, aby před instrukcí `DMAEND` byla ještě instrukce `DMASEV`, která vyvolá přerušení (jeli správně nastaven patřičný registr), které tak může procesoru signalizovat dokončení požadovaného přenosu.

Jakmile je vše připraveno lze zápisem do registru `DBGCMD` spustit DMA program.

2.3.5 Využití rozhraní PRI

PRI rozhraní lze využít mnoha způsoby. Lze jej využít například v situaci, kdy chceme, aby periférie určila počet přenesených dat. Pro dosažení tohoto cíle lze využít instrukcí `DMALPFE` (zahájení cyklu) a `DMALPEND` (ukončení cyklu, pouze v případě nastavení signálu `drlast`). Lze také vhodně využít podmíněné instrukce.

Následuje ukázka jednoduchého programu v jazyce symbolických adres PL330. Předpokládejme čtení z FPGA a zápis do paměti, přičemž na adrese 0x40000000 je řadič generující data, která chceme přenést na adresu paměti 0xFE000000:

```
DMAMOV SAR, 0x40000000  ;nastavení zdrojové adresy
DMAMOV DAR, 0xFE000000  ;nastavení cílové adresy
DMAMOV CCR, SAF DAI     ;zdrojová adresa fixní, cílová inkrementální
            SS4 DS4     ;velikost přenosu 4B
            SB16 DB16   ;délka burstu 16

DMALPFE           ;zahájení cyklu
    DMAWFP 0      ;čekání na request periférie 0, nastaví request_flag

    DMALP 3       ;cyklus o 4 iteracích
        DMALDB    ;podmíněné načtení typu burst
        DMASTB    ;podmíněné uložení typu burst
    DMALPENDB     ;podmíněný skok, program by fungoval i v případě
                  ;nepodmíněného skoku, ale zbytečně by vykonaly DMANOP
    DMALDB        ;poslední načtení typu burst
    DMASTPB       ;poslední uložení typu burst a signalizace periférii

    DMALDS        ;načtení typu single
    DMASTPS       ;uložení typu singu a signalizace periférii
DMALPEND          ;ukončení cyklu, pokud je nastaven signál drlast

DMAWMB  ;čekání na dokončení zápisového přenosu
DMASEV  ;vyvolání přerušení
DMAGO   ;ukončení programu
```

2.4 Linux a ARM architektura

Na procesorech ARM je možné provozovat mnoho operačních systémů. Ať už se jedná o operační systémy reálného času (RTOS[13]), jakými jsou například FreeRTOS nebo RTLinux, nebo systémy obecného použití jako BSD, Windows, či systémy s Linuxovým jádrem. Právě zmíněný Linuxový kernel je podporovaný na mnoha různých typech procesorových architektur. Linux je používaný na velkém množství druhů zařízení a na jeho jádře je založeno mnoho operačních systémů (ačkoliv mnohé z nich nemají slovo Linux v názvu). Jedná se například o ChromeOS, které firma Google nasazuje na zařízeních Chromebook, ale také mobilní systémy jako Firefox OS, Tizen nebo Sailfish OS.

Podpora procesorů ARM v Linuxu prošla postupnými změnami, kdy bylo zpočátku třeba pro téměř každého výrobce trochu odlišnou implementaci, což se negativně projevovalo na rozsahu kódu. Díky politice licencování produktů firmy ARM bylo umožněno výrobcům si své licencované produkty různě konfigurovat a modifikovat. Nebylo tak garantováno, že stejné komponenty použity u různých výrobců budou navzájem kompatibilní – mohly se lišit konfigurace periférií nebo například adresy registrů. V současné době byly tyto problémy z větší části odstraněny, zejména díky využití DeviceTree.

2.4.1 DeviceTree

DeviceTree je stromová struktura používaná k popisu hardware[14]. Snaží se řešit problém, kdy jádro nemá informace o potřebných adresách registrů jednotlivých čipů, jaké zařízení jsou na desce k dispozici atd. Bez DeviceTree by bylo nutné konfigurovat jádro pro každé zařízení nebo desku, a také pro různé, byť podobné, platformy znovu překládat celé jádro. V Linuxu je DeviceTree využíván v architekturách ARM (ale i jinde, například SPARC nebo PowerPC) pro popis hardware, na kterém běží.

Struktura DeviceTree je složená z uzlů (nodes) a vlastností (properties). Každý uzel může být tvořen přímo vlastnostmi zařízení, nebo uzly potomků. Vlastnost tvoří dvojice *název* a *hodnota*. DeviceTree je možné editovat v textovém formátu (přípona `.dts`), ale pro výsledné nasazení se používá jeho binární forma (přípona `.dtb`). Kompilátor DTC[15] umožňuje oba směry převodu mezi textovou a binární formou. Pomocí DeviceTree je možné popsat procesor (počet jader), použitou sběrnici (včetně jednotlivých zařízení, která přes ní komunikují), řadič přerušení a jiné.

2.4.2 Bootování

Bootování na zařízeních typu SoC se může různě lišit podle výrobců. Zde bude popsán proces bootování Linuxových systémů (lze ale nabootovat i samostatnou, jednoduchou aplikaci) na deskách od firmy Xilinx.

Po zapnutí (nebo restartu) desky je prvním místem, odkud se procesor pokusí získat informace pro bootování, BootROM. Jedná se o ROM paměť, která obsahuje kód, podle kterého se procesor rozhodne odkud načte další instrukce pro bootování, tedy zavaděč. Konfiguraci bootování je možné upravit pomocí příslušných pinů na desce. Poté, co je zavaděč (bootloader) načtěn a spuštěn, je třeba nalézt a spusit operační systém. Některé zavaděče, například U-Boot, mají konzoli, díky které je možné manuálně zadávat příkazy,

[13] Real-time operating system
[14] http://www.devicetree.org
[15] Device Tree Compiler

odkud a případně jak má zavést operační systém. Většina moderních zavaděčů umožňuje zavádění systému třeba i po síti.

V případě bootování Linuxu zavaděč načte binární obraz jádra Linuxu (image) a příslušný DeviceTree. U-Boot používá formát obrazu *uImage*, který obsahuje kernel s přidanou hlavičkou obsahující informace pro U-Boot. Souborový systém bývá často uložen jako RAM disk v paměti flash a je použit jako kořenový adresář Linuxového systému. Dalším krokem je načtení DeviceTree blobu, který je později použit jádrem pro získání informací o hardwaru, s kterým bude jádro komunikovat. Po úspěšném načtení všech částí nutných pro běh operačního systému je Linux spuštěn a je mu předáno řízení procesoru. Díky popisu v DeviceTree může Linux automaticky načíst ovladače pro dostupný hardware.

2.5 Tvorba ovladačů v OS Linux

Tvorba ovladačů pro Linux (nebo analogicky pro jiné jádro) vyžaduje hlubší znalost nejen programovacího jazyka, ale i hardware, pro který je ovladač vyvíjený a hlavně znalost Linuxu samotného. Jelikož se zdrojové kódy kompilují vůči konkrétní verzi jádra, je před samotným vývojovým procesem nutné nakonfigurovat Linuxové jádro a zkompilovat jej. Protože moduly jádra jsou spouštěny přímo ve vrstvě samotného jádra (narozdíl od user space aplikací – tedy aplikací běžících v uživatelském prostoru), je nutné počítat s tím, že spouštění takových modulů může vážně ohrozit stabilitu testovacího počítače – chybnou manipulací s jadernými prostředky může jádro havarovat (poté je zpravidla nutné restartovat celý systém).

Linuxové jádro vývojářům modulů poskytuje API[16]. Bohužel ne všechny funkce jádra jsou pečlivě dokumentovány, a tak je někdy nezbytné získání informací pročítáním zdrojových kódů. Pro procházení zdrojových kódů jádra a jeho historie lze využít git[17] a pro snadné vyhledání identifikátorů například webovou stránku *free-electrons*[18]. Dokumentace funkcí a struktur se zapisuje do kódu ve formátu `kernel-doc`, který byl vyvinut pro Linuxové jádro[8]. Další dokumentaci je možné nalézt v adresáři `Documentation`.

2.5.1 Programovací techniky

Odlišností od programování aplikací v uživatelském prostoru je především fakt, že zde chybí standardní knihovny jazyka C. Jádro poskytuje své vlastní knihovny. Například místo standardní funkce `printf()` (formátovaný tisk na standardní výstup) lze použít podobnou funkci poskytovanou jádrem – funkci `printk()`. Funkce `printk()` navíc specifikuje `loglevel`, který určuje, o jaký typ zprávy se jedná (například `KERN_WARNING` pro varovné zprávy nebo `KERN_INFO` pro informační). Záleží na typu zprávy a nastavení Linuxové distrubuce, kam se tato zpráva vytiskne a zaloguje.

Při vývoji v jaderném prostoru je neustále nutné mít na vědomí fakt, že chybná manipulace s pamětí může způsobit nestabilitu systému (neuvolněná paměť zůstane nevyužitelná do restartování systému a přístup na špatnou adresu může způsobit havarování celého systému). Je také nutné dbát na zpracování návratových kódu volaných funkcí. Pro řešení obsluhy chyb se v jádře používá příkaz `goto` a práce s návěštími. Protože moderní operační systémy uvolňují paměť po ukončení uživatelské aplikace a hlídají přístupy do paměti, je

[16] Application Programming Interface
[17] https://git.kernel.org/cgit/
[18] http://lxr.free-electrons.com/

obzvláště důležité uvědomit si absenci takovýchto mechanismů při vývoji jádra a brát v potaz případné následky.

Počet řádků zdrojových kódů Linuxu v dnešní době přesahuje hodnotu 15 milonů[5]. Pro zjednodušení práce se používá systém *Kbuild* (tento systém používá také Coreboot), který (pomocí programu *make*) překládá jednotlivé zdrojové kódy modulů. *Kbuild* podle zadané cesty k adresáři jádra zjistí potřebné informace a vytvoří výsledný modul (kernel module, přípona .ko).

2.5.2 Adresový prostor

Při práci s jádrem Linuxu se rozlišují dva druhy paměťových prostorů: uživatelský (user space) a jaderný (kernel space). Uživatelský paměťový prostor je používán v běžných (uživatelských) aplikací, zatímto jaderný prostor v modulech jádra[3]. Linuxové jádro totiž využívá privilegovaného režimu, který řeší některá bezpečností rizika (aplikace v uživatelském prostoru má omezený přístup k hardwaru). Některé procesory (například ARM[19]) rozlišují dvě úrovně – privilegovaný a neprivilegovaný režim, procesory x86 znají dokonce čtyři režimy, takzvané ringy. Zatímco v privilegovaném režimu běží Linuxové jádro (kernel space), tak v neprivilegovaném běží aplikace (user space). Pro kopírování dat mezi těmito adresovými prostory poskytuje jádro funkce `copy_to_user()` – pro kopii z jaderného prostoru do uživatelského, a `copy_from_user()` naopak.

Jako operační systém s podporou virtuální paměti rozlišuje Linux několik typů adres paměti[3]:

- **Uživatelská virtuální adresa** - používá se pro adresování v aplikacích v uživatelském prostoru.
- **Fyzická adresa** - Adresa používaná při komunikaci mezi procesorem a pamětí.
- **Adresa sběrnice** - Adresa používaná při komunikaci mezi periférními zařízeními a pamětí. Využívá ji například jednotka DMA.
- **Logická jaderná adresa** - Adresování používáno jádrem. Logické adresy jsou mapovány jedna ku jedné vůči fyzickým adresám. Lze alokovat funkcí `kmalloc()`.
- **Virtuální jaderná adresa** - Virtuální adresování, které přímo neodpovídá fyzická adresa. Lze alokovat pomocí funkce `vmalloc()`.

2.5.3 Moduly

Moduly jádra jsou objektové soubory, které umožňují rozšířit funkcionalitu operačního systému například o podporu nového hardwaru. V případě Linuxu lze moduly zkompilovat spolu s jádrem, nebo je možné je načíst za běhu systému (tuto funkcionalitu neposkytují všechny operační systémy, například OpenBSD ji odstranil v srpnu roku 2014[20]). Zejména pro načítání modulů za běhu, kdy je možnost, že bude načítán modul kompilovaný pro jinou verzi jádra, jsou důležité informace o kontrolních součtech jádra uvnitř modulu. Díky těmto informacím je možné zjisit, zdali zaváděný modul má kompatibilní ABI jádra s jádrem systému.

[19] http://wiki.osdev.org/ARM_Overview
[20] http://www.openbsd.org/faq/current.html#20141013

Moduly, narozdíl od většiny aplikací v jazyce C, neobsahují funkci `main()`. Naproti tomu moduly obsahují několik vstupních bodů (funkcí), které jádro využije při natažení modulu (`module_init()`) nebo zrušení modulu (`module_exit()`). Pro natažení modulu slouží aplikace `insmod` a `modprobe`. Modprobe je pokročilá aplikace, která navíc řeší závislosti modulů. Obdobně pro odstranění modulu se použijí aplikace `rmmod` a `modprobe` (s parametrem -r). Aktuálně natažené moduly lze zobrazit příkazem *lsmod*.

Dále moduly zpravidla používají několik maker. Jedná se například o makro pro nastavení autora daného modulu `MODULE_AUTHOR`, makro `MODULE_DESCRIPTION` pro popis modulu a `MODULE_LICENSE`[6]. Poslední zmíněné makro je obzvláště důležité, neboť určuje pod jakou licencí je modul šířen. Tato informace se popužívá při poskytování API jádrem. Jádro totiž poskytuje jen takové funkce, které jsou označeny pomocí makra `EXPORT_SYMBOL` (jak ale ukázaly některé sotwarové firmy, tak lze toto omezení obejít[21]).

2.5.4 Znaková zařízení

Znaková zařízení (char devices) jsou speciální soubory (lze s nimi provádět obvyklé operace jako čtení a zápis), které spravuje jaderný modul (který byl k takovému zařízení přiřazen, nebo jej vytvořil). Znaková zařízení se nacházejí v adresáři */dev*. Kromě znakových zařízení existují ještě například bloková zařízení, ke kterým je přistupováno po blocích dat. Typ zařízení lze zjistit pomocí příkazu *ls -s*, kde první sloupec výstupu značí typ zařízení (c = znakové, b = blokové).

Každému zařízení jsou přiřazena *MAJOR* a *MINOR* čísla. *MAJOR* čísla identifikují ovladač, který je s daným zařízením spojen. *MINOR* čísla označují samotná zařízení[3]. Modul pro obsluhu zařízení definuje funkce, které zařízení bude podporovat. Jedná se například o funkce *open()* a *close()* při otevření a zavření souboru a funkce *read()* a *write()* při čtení a zápisu. Zápis 10 bajtů ze souboru *input* do imaginárního zařízení *char* se provede například takto:

```
head -c 10 input > /dev/char
```

Funkce volané operačním systémem pro toto zařízení jsou volány v následujícím pořadí: *open()*, *write()*, *close()*.

Operační systém automaticky otevře zařízení při prvním přístupu a zavře ho až při posledním přístupu. Pokud zařízení není připraveno zpracovávat data, tak může informovat systém patřičnou návratovou hodnotou.

2.5.5 Ovladače zařízení

Programování ovladačů zařízení je mírně odlišné od modulů popsaných v kapitole 2.5.3. Jádro poskytuje pro práci s ovladači speciální API. Hlavičkový soubor `device.h` poskytuje makro

```
module_driver(__driver, __register, __unregister, ...)
```

které nastaví funkci `__register()` a `__unregister()`, jenž budou zavolány při registraci, respektive zrušení, modulu se strukturou `__driver` jako parametrem. Struktura `__driver` obsahuje, mimo jiné, ukazatele na funkce jako `probe()` a `remove()`, strukturu `of_device_id`, jméno a nebo vlastníka.

Linuxové jádro je schopné určit, jaký ovladač přiřadit ke kterému zařízení. Aby toto bylo možné, je potřeba využít informace z DeviceTree. DeviceTree totiž kromě informací o adrese registrů zařízení na sběrnici, počtu přerušení atd. lze využít také k vyhledání

[21] http://lwn.net/Articles/82306/

vhodného ovladače. Zařízení může obsahovat mimo samotný název i vlastnost *compatible*, která může být využita při vyhledávání ovladače s odpovídající kompatibilitou (zadanou pomocí prvku `.compatible` ve struktuře `of_device_id` při registraci ovladače). Poté, co jádro nalezne příslušný ovladač, je vyvolána funkce `probe()`. Do této doby byl ovladač zařízení neaktivní. Obdobně při odebrání zařízení je zavolána funkce `remove()`.

2.6 DMA v Linuxu

Linux poskytuje hned několik API pro práci s DMA. Jsou tu například specifická API pro sběrnice ISA a PCI, ale také obecnější jako je DMA Engine. Bohužel samotná dokumentace Linuxu k DMA poskytuje pouze strohé informace k pokrytí problematiky jednotlivých rozhraní.

2.6.1 Linuxové rozhraní DMA Engine

DMA Engine je Linuxové API pro DMA přenosy, které poskytuje vrstvu nezávislou na architektuře a sběrnici. Podporuje základní typy přenosů jakými jsou paměť–paměť a paměť–zařízení, dále konfiguraci kanálu atd. Toto API stále prochází vývojem, ačkoliv stále není zdaleka vše patřičně zdokumentováno (při bližším zkoumání tohoto API je zřejmé, že pro dosažení stejného výsledku lze využít více cest).

Následuje kostra aplikace, která přenese data z paměti `src_buf` do paměti na adrese `dst_buf`:

```
/* Nastavení masky DMA */
dma_cap_zero(mask);
dma_cap_set(DMA_MEMCPY,mask);

/* Získání DMA kanálu */
dma_chann = dma_request_channel(mask, 0, NULL);

/* Mapování paměti -- získání adresy sběrnice */
dst_bus_addr = dma_map_single(dma_dev->dev, dst_buf, bufsize, DMA_FROM_DEVICE);
src_bus_addr = dma_map_single(dma_dev->dev, src_buf, bufsize, DMA_TO_DEVICE);

/* Nastavení přenosu a zařazení požadavku do fronty */
tx_desc = dma_dev->device_prep_dma_memcpy(dma_chann,
    dst_bus_addr, src_bus_addr, buf_len, flags);
cookie = dmaengine_submit(tx_desc);

/* Aktivace přenosu */
dma_async_issue_pending(dma_chann);
```

Toto byl zjednodušený příklad. Pro signalizaci dokončení přenosu lze využít callback funkce a synchronizační mechanismy, jakým je například *Completion*[22], které poskytuje Linux API.

[22] Jedná se o mechanismus, který umožňuje jednoduše signalizovat dokončenou úlohu mezi vlákny

2.6.2 Ovladač PL330

Ovladač jednotky DMA PL330 má za sebou několikaletý vývoj. Ve dřívějších verzích upraveného Linuxového kernelu firmou Xilinx byl přítomen ovladač PL330 vyvinutý právě firmou Xilinx. Tento ovladač byl modelován na API sběrnice ISA (ačkoliv PL330 používá sběrnici AXI) a umožňoval DMA přenosy mezi zařízením a pamětí. Ovladač podporoval také přenosy typu burst. V roce 2010 firma Samsung vyvinula nový ovladač pro jednotku PL330, který již využívá obecnější Linuxové API pro přenosy DMA – DMA Engine. Ačkoliv tento ovladač nepodporuje burst přenosy, stal se součástí Linuxového kernelu a také upraveného kernelu firmy Xilinx.

Rozdíl mezi těmito dvěma ovladači je tedy především v tom, jaké kernelové rozhraní je použito a které přenosy jednotlivé ovladače podporují. Je také velký rozdíl v dokumentaci obou ovladačů. Ovladač firmy Xilinx se jeví jako jednodušší (přehlednější) a lépe zdokumentovaný, naproti tomu novější ovladač je zdokumentovaný velmi stručně. Výhodou ovladače od Xilinxu je také fakt, že programátorovi nabízí prostředky pro použití vlastního programu pro jednotku DMAC.

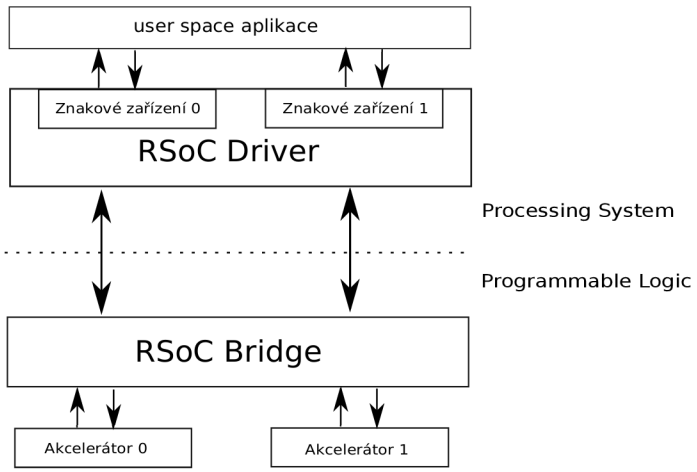
2.7 RSoC Framework

Komponenty SoC mají výhodu ve vysokém výkonu díky optimalizovanému hardwaru na konkrétní úkolu. Nevýhodou SoC ale je, že může být použit pouze na jediný typ úlohy. Vedle SoC existuje ještě RSoC[23], které disponují navíc jednotkou FPGA. Lze stejně tak optimalizovat úlohu pro konkrétní aplikaci, ale tento čip je dále znovupoužitelný. V případě potřeby jej tak lze upravit pro jiný typ aplikace. To má výhodu i v tom, že lze novou jednotku navrhnout v relativně krátkém čase a ihned ji otestovat (a využít tak zpětné vazby k případné úpravě návrhu).

V současné době je vyvíjen framework, který nabízí univerzální přístup k vývoji aplikací právě na platformě RSoC – jedná se o RSoC Framework. RSoC Framework tvoří rozhraní mezi PS[24] (procesor) a PL[25] (například FPGA) [7]. Z pohledu RSoC Frameworku je totiž hlavní jednotkou procesor a FPGA tvoří jen další periférii (ovšem konfigurovatelnou).

RSoC Framework zahrnuje hned několik částí. Jednou z nich je i RSoC Accelerator, který nemusí nutně něco akcelerovat (urychlovat), ale je definován spíše jako rozhraní (může se jednat o jakoukoliv hardwarovou jednotku). Další částí je RSoC Bridge, který propojuje akcelerátory s procesorovou jednotkou a tvoří mezivrstvu mezi softwarem a hardwarem. RSoC Framework dále poskytuje interface pro user space aplikace – RSoC Drivers (například pro systémy Linux a FreeRTOS). RSoC Drivers poskytují přístup k akcelerátorům skrze znaková zařízení (tedy hlavně operace čtení a zápis). Komunikace mezi akcelerátorem a user space aplikací pomocí znakových zařízení je znázorněna na obrázku 2.3. Za zmínku stojí, že RSoC Acceleratory komunikují na bázi rámců (frames)[26].

[23] Reconfigurable System-on-Chip
[24] Processing System
[25] Programmable Logic – programovatelnou logikou
[26] rámce se skládají z 4B hlavičky určující velikost rámce a samotného těla rámce



Obrázek 2.3: RSoC Framework

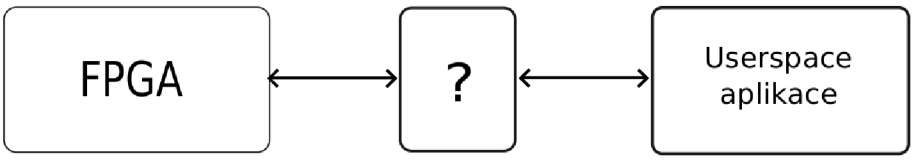
Kapitola 3

Rozbor problému a řešení

V této kapitole je popsán rozbor problému a nástin komunikace mezi user space aplikací a FPGA jednotkou.

3.1 Rozbor problému

Jak je z obrázku 3.1 patrné, bylo třeba navrhnout komunikační jednotky v rámci FPGA a implementovat softwarovou aplikaci (aplikace), které by zprostředkovaly patřinou komunikaci mezi user space aplikací a FPGA akcelerátorem. Zadáním je využít jednotku PL330. Jelikož je tento DMA obvod programovatelný (disponuje několika vlastními instrukcemi), je možné sestavit program přizpůsobený konkrétní aplikaci. Nabízí se také možnost využít Peripheral request interface, kterých mají platformy Xilinx Zynq a Altera SoC shodně čtyři. Jelikož RSoC Accelerátory pracují s rámci, tak je nutné, aby jednotka PL330 uměla rozlišit jednotlivé rámce (detekovat konce rámců) a pracovat s nimi.



Obrázek 3.1: Zadání

Protože Linuxové ovladače pracují v kernel space (ovladač PL330 není výjimkou), ale zároveň je potřeba obsluhovat uživatelské (user space) požadavky, bylo řešení rozděleno na dvě části.

První částí je ovladač jednotky PL330. Tento ovladač má na starosti obsluhu dostupných DMA kanálů jednotky PL330, sestavení PL330 programu podle požadavků (případně poskytnout rozhraní pro vlastní návrh PL330 programu), obsluhu přerušení, nastavování registrů a jiné. Předpokládá se, že FPGA jednotka bude obsahovat několik akcelerátorů, které budou dodávat, případně sbírat, data do DMA jednotky. Pro každý jednotlivý akcelerátor budou alokovány dva DMA kanály – jeden pro čtení z akcelerátoru a druhý pro zápis do něj. Tento DMA ovladač (prostřednictvím PL330 jednotky) bude tedy do přiřazených akcelerátorů zapisovat a číst data a předávat je do další vrstvy.

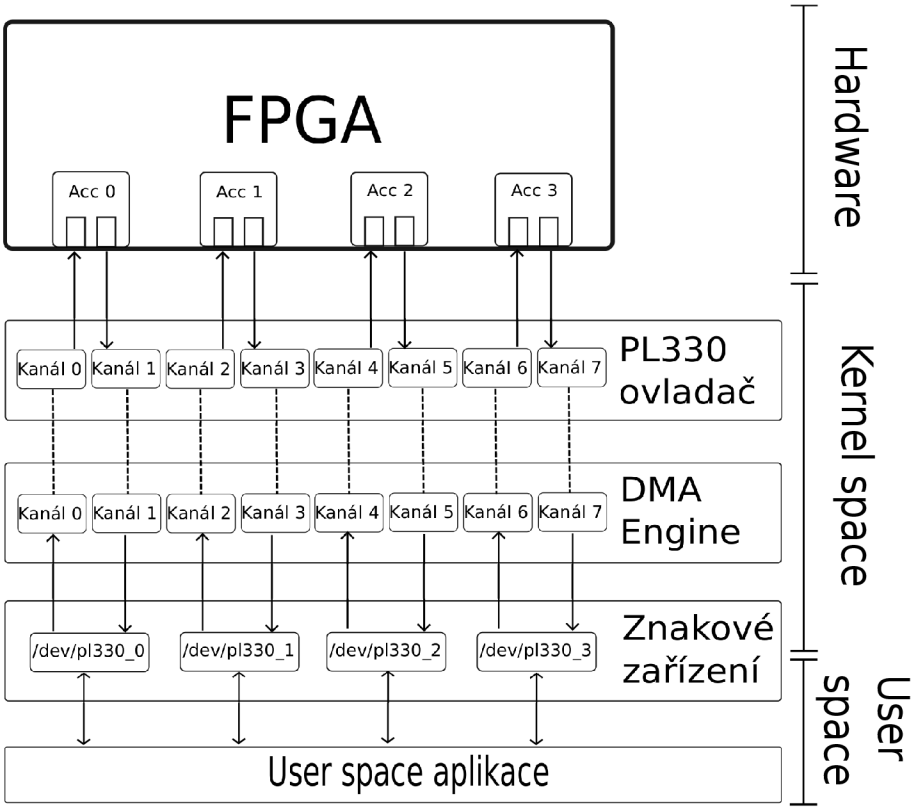
Druhou částí, vrstvou nad DMA ovladačem, je ovladač znakových zařízení. Tento ovladač bude poskytovat pro každý akcelerátor jedno znakové zařízení (char device). Pro jed-

notku PL330 disponující osmi DMA kanály (například platforma Xilinx Zynq) a pro dostupné čtyři akcelerátory bude tedy ovladač znakového zařízení poskytovat tyto jednotky:

- **/dev/pl330_0**
- **/dev/pl330_1**
- **/dev/pl330_2**
- **/dev/pl330_3**

Ovladač znakového zařízení rozpozná žádost o čtení a zápis z jednotlivých znakových zařízení, provede kopii dat mezi user space a kernel space a vyžádá si přenos dat konkrétním DMA kanálem.

Poslední částí řešení bylo navrhnout způsob komunikace mezi ovladačem znakového zařízení a ovladačem PL330 jednotky. K tomuto účelu bylo zvoleno DMA Engine API, které poskytuje rozhraní pro alokaci a uvolnění libovolných kanálů (při alokaci lze vyfiltrovat kanál s podporou PRI), nastavení parametrů přenosu, jeho vykonání atd. DMA Engine je tedy mezivrstvou mezi jednotlivými ovladači. PL330 ovladač skrze DMA Engine registruje služby (funkce), kterých může využít libovolný další modul (v tomto případě ovladač znakového zařízení). Výsledné řešení je zobrazeno na obrázku 3.2.



Obrázek 3.2: Návrh řešení

Komunikace mezi FPGA jednotkou a ovladačem PL330 je popsána v kapitole 3.2. Vlastní implementaci softwarové části řešení (ovladač PL330, znakové zařízení a user space aplikace) je věnována kapitola 4.

3.2 Návrh komunikace

Tato kapitola popisuje komunikaci mezi jednotkou FPGA a PL330. Nutnou podmínkou řešení byla podpora detekce a signalizace konce rámce na straně DMA, aby DMA jednotka byla schopna přenést značku konce rámce i do dalších vrtev komunikace.

3.2.1 Nástin čtení z FPGA s využitím PRI

Při přenosu dat z FPGA do DMA jednotky je základním problémem skutečnost, že dopředu není známka délka dat (a tedy ani jak velké budou muset být jednotlivé buffery). Pro detekci konce rámce se zdálo být vhodné využít kanálu PRI, který obsahuje signál `drlast` (tento typ signálu je veden pouze v jednom směru, jak je patrné z ilustrace 2.2, takže jej není možné využít pro signalizaci konce rámce).

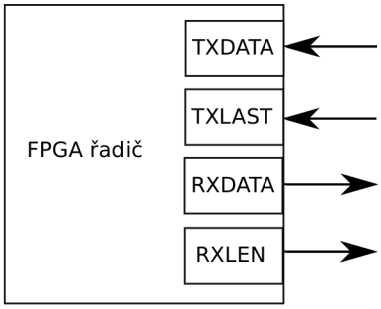
Z hlediska úspory zdrojů a počtu přenešených dat by bylo nejvhodnější, pokud by DMA jednotka detekovala konec rámce (například pomocí signálu `drlast`, který se dá využít pro ukončení cyklu, jak je popsáno v sekci 2.3.2). DMA by tak mohlo číst data (nejlépe v nějakém jednoduchém cyklu), dokud by nebyl detekován konec rámce, nebo dokud by nebyly naplněny buffery. Bohužel má ale PL330 v tomto směru velmi omezenou instrukční sadu, a tak není možné kombinovat ukončení cyklu pevným počtem iterací (tedy klasický cyklus `DMALP`) a ukončení signálem `drlast` (tedy nekonečný cyklus `DMALPFE`).

Pro signalizaci konce rámců je potřeba hledat jinou cestu. Pro řešení problému byla navržena fronta obsahujícího délky jednotlivých rámců. FPGA řadič při vydávání dat počítá přenášené bajty a v případě, že je signalizován konec rámce, bude délka tohoto rámce uložena ve frontě délek. DMA proto kromě čtení samotných dat z určeného registru musí také číst frontu délek rámců, což je bohužel další režie navíc. DMA jednotka tedy po naplnění bufferu přenese data a poté ještě přenese délky rámců. Aby nedošlo ke zbytečnému čekání při přenosu dat, pokud by žádná data nebyla zrovna k dispozici, tak byla zvažována možnost, že by v takovém případě FPGA signalizovalo `no data` (žádná data). Bohužel ale nebylo nalezeno řešení, při kterém by bylo možné bezpečně rozlišit, zdali přenos dat skončil kvůli přečtení celého bufferu nebo pro signalizaci konce dat. Ničemu nepomůže ani získání poslední cílohé adresy dat PL330 kanálu (viz registr `DA` v sekci 2.3.3), protože tato adresa je přepsána cílovou adresou pro délky rámců.

Existuje ještě možnost podmíněného přenosu délek rámců. Lze k tomu využít podmíněné instrukce, které podle signálu `drtype` provádí buď single, nebo burst přenosy. FPGA by tak pomocí `drtype` signalizovalo, zdali se nachází ještě nějaká data ve frontě délek rámců. Zásadní nevýhodou tohoto řešení by byl fakt, že by se jeden z přenosů dat a délek rámců musel přenášet pomocí burst přenosů a druhý pomocí single. Právě pro nutnost použít single přenosy byla zavržena i tato možnost.

3.2.2 Zvolené řešení

Konečné řešení je komplikovanější, a to z důvodu omezené PL330 instrukční sady. Kromě přenosu samotných dat je nutné přenést i délky rámců (je to další režie a navíc si tato fronta bere nějaké zdroje FPGA). Příjímání a odesílání rámcu uvnitř FPGA zajišťuje řadič, jehož adresový prostor je zobrazen na obrázku 3.3. `TXDATA` a `RXDATA` slouží k vydávání dat z řadiče, respektive ke čtení dat. Registr `TXLAST` slouží k signalizaci konce rámce a fronta `RXLEN` vydává délky přečtených rámců. FPGA řadič dále disponuje timeoutem, který vyvolá chybu na sběrnici, pokud nejsou po určitu dobu přítomna žádná data. Lze se tak vyvarovat určitému zamrznutí DMA kanálu v případě, že není naplněn celý buffer, ale žádná data



Obrázek 3.3: FPGA řadič

dlouho nepřichází.

Čtení dat z FPGA (RX přenos)

Pro čtení dat z FPGA dává řadič k dispozici dvě fronty - jedna fronta pro data (`RXDATA`)a jedna pro délky rámců (`RXLEN`). Software musí zařídit, aby se nepřečetlo více rámců, než kolik se jich vejde do fronty délek - může se tedy stát, že bude nutné číst buffery po částech. Počet bajtů z `RXDATA`, které lze bezpečně přenést aniž by přetekla fronta délek rámců, je vyjádřen následujícím vztahem:

$$pocet\_bytu = hloubka\_delek * min\_ramec$$

kde `hloubka_delek` je velikost (hloubka) fronty délek rámců (hodnotu lze vyčíst z FPGA) a `min_ramec` je minimální délka jednoho rámce - jedná se o předem domluvenou konstantu (obvykle 16).

Jelikož se fronta délek rámců čte v každém případě celá (tedy i v případech, pokud fronta není plná nebo je dokonce zcela prázdná), je nutné poskytnout vždy celou frontu k přečtení. FPGA po každém vyprázdnění fronty data nuluje, díky čemuž může softwarový ovladač rozpoznat, kolik délek rámců je skutečně k dispozici. Nepředpokládá se výskyt rámců nulové délky, neboť každý rámec kratší než minimální povolená hodnota je považován za chybný.

Zápis dat do FPGA (TX přenos)

Podobně jako pro RX přenos, tak i pro TX poskytuje FPGA řadič dva registry: jeden pro přenos samotných dat (`TXDATA`) a druhý pro signalizaci konce rámce (`TXLAST`). Zápisem patřičné hodnoty do `TXLAST` se FPGA řadiči signalizuje, že **následující** burst je poslední v daném rámci (a také se tím určí, kolik bajtů je platných v posledním přeneseném slově). `TXLAST` může tedy nabývat následujících hodnot (jedná se opět o 32 bitový registr):

- 0x0001 - poslední přenesené slovo má platný první bajt.
- 0x0011 - poslední přenesené slovo má platné první dva bajty.
- 0x0111 - poslední přenesené slovo má platné první tři bajt.
- 0x1111 - poslední přenesené slovo je platné v celém rozsahu (4 bajty).

Klasický TX přenos rámce rozděleného do `n` burstů bude tedy probíhat tak, že se přenese `n - 1` burstů rámce do registru `TXDATA`, poté se signalizuje počet platných bajtů v

posledním slově posledního burstu do registru `TXLAST` a nakonec se přenese poslední datový burst do registru `TXDATA`.

Zotavení se z chyb

FPGA řadič může signalizovat dva typy chyb:

- chyba v datech (například špatná velikost rámce)
- vypršení timeoutu (nebyla přítomna žádná data po určitou dobu)

PL330 jednotka má pouze jeden bit pro signalizaci stavu sběrnice - stav OK a chybový stav (PL330 tedy nerozlišuje, o jaký typ chyby na AXI sběrnici se jedná).

Řadič tak v případě chyby nebo vypršení timeoutu signalizuje chybu na sběrnici, kterou detekuje manager vlákno jednotky PL330 a ukončí vlákno příslušného DMA kanálu (takto PL330 reaguje na každou chybu).

Problém nastává při chybě přenosu uprostřed burstu. Načtení a uložení dat jsou v PL330 samostatné instrukce, a tak v případě chyby při načtení je vyvoláno přerušení manager vláknem a instrukce pro uložení již není zpracována. Nastává tak situace, kdy data nejsou ani v FPGA (byla načtena burst přenosem), ani přenesena na cílovou adresu (instrukce pro uložení dat není vykonána) - data jsou totiž ve vnitřním buffer PL330 jednotky, přičemž již není možné je odtud nikam přenést (kanál se nachází v chybovém stavu). V případě chyby v datech to takový problém není, protože s danými daty nebude dále pracováno. Pokud ale bude vyvolána chyba při vypršení timeoutu, je nezbytné zajistit přenesení všech platných dat. Proto bude nutné do řadiče FPGA implementovat pomocný buffer, který bude uchovávat hodnoty přenesené posledním burstem pro případ chyby. V takovém případě by měl být ovladač schopen chybu detekovat a dodatečně přenést načtenou část burstu.

Kapitola 4

Softwarová implementace

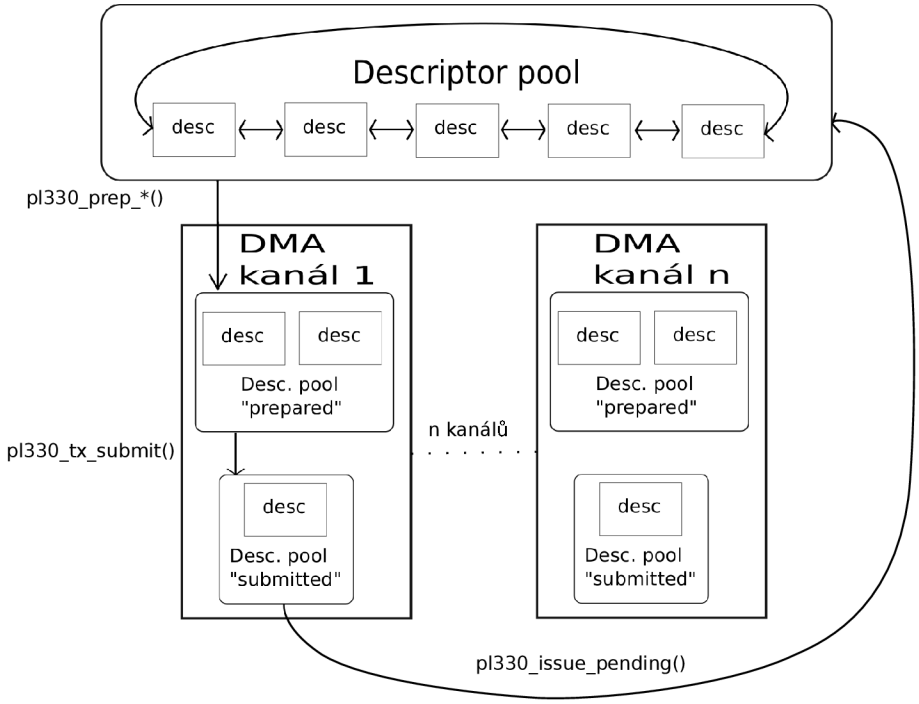
V této kapitole bude podrobně rozebrána softwarová implementace řešení. Softwarové řešení se skládá ze dvou sammostatných Linuxových modulů a jedné user space aplikace. Veškerý zde popsán software byl naprogramovaný v jazyce C.

4.1 Ovladač PL330

Ovladač PL330 jednotky je jaderný modul navržený pro potřebny řešeného problému. Ovladač vychází ze dvou existujících ovladačů, a to mainstreamého ovladače od firmy Samsung a ovladače firmy Xilinx (již nevyvíjený). Jelikož ovladač firmy Xilinx není součástí kernelového upstreamu (a tedy není dále vyvíjen) a nový ovladač nepodporuje přenosy typu burst (navíc nenabízí možnost použít libovolný PL330 program), tak se nejevý ani jeden z ovladačů jako vhodný kandidát pro použití k řešení problému. Bylo proto nutné vyvinout vlastní ovladač této jednotky.

Linuxové ovladače implementující rozhraní DMA Engine využívají pro popis přenosu DMA descriptory. Nejinak se chová i ovladač PL330, který si descriptory přenosů uchovává v kruhovém seznamu (je k tomu využit datový typ `list_head` poskytovaným rozhraním jádra). Jak je vidět na ilustraci 4.1, kanál si v případě potřeby vezme volný descriptor ze seznamu a po dokončení přenosu jej vrátí zpět. Obslužný modul pomocí funkce `pl330_prep_dma_memcpy()` získá připravený descriptor (a kanál si takový descriptor alokuje ze seznamu descriptorů), poté může modul tomuto descriptoru nastavit některé atributy (například callback funkci po dokončení přenosu a její parametry) a přidat daný descriptor do čekací fronty (pomocí funkce `pl330_tx_submit()`). Nakonec je po zavolání `pl330_issue_pending()` zahájena obsluha přenosu. Po dokončení přenosu je zavolán příslušný callback (jeli nastaven) a descriptor je vrácen zpět do seznamu. Pokud nejsou žádné descriptory k dispozici, ovladač za běhu vytvoří nové.

Aby byly funkce pro zápis PL330 instrukcí poskytované ovladačem co nejblíže syntaxi jazyku symbolický adres PL330, byl do ovladače implementován jednoduchý překladač těchto funkcí do strojového jazyka jednotky PL330. Ne všechny PL330 instrukce jsou totiž skutečnými instrukcemi. Například instrukce `DMALPFE` neprovádí žádnou operaci, ale slouží pouze ke sdělení překladači, že příští instrukce `DMALPEND` bude ukončovat nekonečnou smyčku a na jakou adresu bude proveden skok. Součástí kódu instrukce `DMALPEND` je totiž offset, o který se má programový čítač posunout zpět. Vyvinutý překladač si tak při instrukci `DMALP` a `DMALPFE` zapamatuje aktuální adresu programu, a pro instrukci `DMALPEND` tuto adresu vloží do strojového kódu. Tyto informace pro překladač se uchovávají ve struktuře, která se pře-



Obrázek 4.1: Description pool

dává jako parametr funkcím odpovídajícím PL330 asembleru. Následuje ukázka programu zapsaného pomocí PL330 syntaxe:

```
DMAMOV SAR, 0x40000000
DMAMOV DAR, 0xFE000000
DMALP 8
    DMALD
    DMAST
DMALPEND

DMAWMB
DMASEV 0
DMAGO
```

Pro sestrojení výše uvedeného PL330 programu v jazyce C bude nutné volání následující posloupností funkcí poskytovaných ovladačem PL330:

```
DMAMOV(&prog, PL330_SAR, 0x40000000);
DMAMOV(&prog, PL330_DAR, 0xFE000000);
DMALP(&prog, 8);
    DMALD(&prog);
    DMAST(&prog);
DMALPEND(&prog);

DMAWMB(&prog);
DMASEV(&prog, 0);
DMAGO(&prog);
```

Jelikož požadavky na PL330 program mohou být specifické pro konkrétní úkol a všechny možnosti konfigurace DMA přenosu pomocí rozhraní DMA Engine nelze pokrýt, poskytuje ovladač PL330 funkci pro vložení ručně napsaného programu. Vedle standardní funkce `pl330_prep_dma_memcpy()` (která je volána skze funkci `device_prep_dma_memcpy()` rozhraní DMA Engine) poskytuje ovladač navíc funkci `pl330_prep_program_addr()`, pomocí které je možné zadat adresu vlastního PL330 programu.

4.2 Ovladač znakových zařízení

Ovladač znakových zařízení je jaderný modul spolupracující s ovladačem PL330. Ovladač znakových zařízení při inicializaci zjistí informace z řadiče FPGA (počet akcelerátorů, typ akcelerátorů, jejich adresa atd). Pro podporované akcelerátory ovladač vytvoří znakové zařízení a pro každé z nich alokuje dva DMA kanály pro čtení a zápis dat (tyto kanály jsou alokovány z PL330 ovladače skrze DMA Engine). Pokud je akcelerátor potřeba inicializovat, je tak učiněno ještě před přidáním patřičného znakového zařízení do systému. Je vytvořeno jen tolik znakových zařízení, pro kolik lze alokovat DMA kanály (pro platformu Xilinx Zynq s 8 kanály lze tedy vytvořit maximálně 4 znaková zařízení). Znakové zařízení má název `pl330_n`, kde `n` je číslo znakového zařízení.

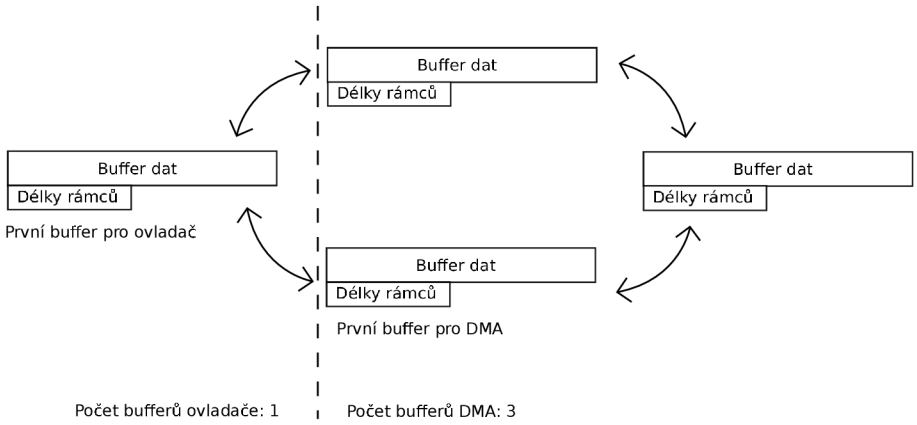
Po inicializaci a přidání znakového zařízení do systému je ovladač připraven pro zpracování požadavků user space aplikací. Čtení 2048 bajtů z akcelerátoru lze provést příkazem:

```
head -c 2048 /dev/pl330_0
```

Pro zapsání rámce délky 512 bajtů (uložených v souboru *input*) do akcelerátoru lze provést následovně:

```
head -c 512 input > /dev/pl330_0
```

Ovladač znakových zařízení používá pro ukládání a načítání dat z akcelerátorů kruhový seznam bufferů (pro každý DMA kanál). Tento seznam bufferů je sdílený pro ovladač a DMA kanál jednotky PL330 – je tak nezbytný mechanismus, který zajistí aby k jednomu bufferu přistupoval pouze ovladač nebo DMA kanál (ne oba zároveň). Ovladač proto pro každý kanál udržuje ukazatele na první buffer alokovaný pro ovladač a počet bufferů, a také první buffer alokovaný pro DMA kanál a počet těchto bufferů. Obrázek 4.2 demostruje příklad čtyř bufferů, kdy jeden je alokován pro ovladač a tři zbývající pro DMA kanál.



Obrázek 4.2: Kruhový seznam bufferů

Kanál pro čtení dat z FPGA řadiče je spuštěn ihned po inicializaci zařízení. Jakmile DMA kanál dokončí přenos dat (naplní buffer), je vyvoláno přerušení. V obsluze tohoto

přerušení ovladač znakových zařízení uvolní zaplněný buffer pro použití ovladačem, posune ukazetel prvního bufferu DMA kanálu na další buffer a zahájí nový přenos. Pokud přijde user space požadavek na čtení dat, začnou se postupně vyprazdňovat zaplněné buffery a po úplném vyprázdnění je takový buffer uvolněn pro použití DMA kanálem.

Zápis probíhá opačným způsobem. Nejprve jsou buffery naplňované daty z user space požadavkem *write()* a po úplném zaplnění je vytvořen PL330 program, který přenese data (včetně signalizací délek rámce) do řadiče FPGA.

Zápis a čtení z akcelerátorů se neprovádí vždy při operaci *write()*, případně *read()*, ale až po naplnění celého bufferu daty z user space (v případě zápisu do akcelerátoru), nebo po naplnění celého bufferu daty z akcelerátoru (v případě čtení z akcelerátoru). Před každým přenosem dat mezi ovladačem znakových zařízení a FPGA řadičem je sestavován PL330 program s využitím exportovaných funkcí ovladače PL330.

4.3 User space aplikace

Pro čtení a zápis dat do znakových zařízení postačuje funkcionalita Unixové aplikace head s parametrem `-c` (vypíše určitý počet bajtů), která je také implementovaná v programu BusyBox (BusyBox je aplikace, která má v sobě vestavěné implementace standardních Unixových příkazů, včetně *head* nebo *tail*). Busybox implementace příkazu head má ale nežádoucí chování při operaci *read*, kdy má head požadavek pro přečtení velikost jedné stránky paměti (obvykle 4KiB). Může tak nastat situace, kdy je zažádáno o přečtení například 128 bajtů dat, ale příkaz head přečte 4KiB a vypíše 128 bajtů. V takovém případě jsou zbývající data přečtena, ale nepoužita.

Pro komunikaci se znakovými zařízeními tak byla implementována i jednoduchá user space aplikace založená na aplikaci head. Narozdíl od Unixové implementace tato aplikace podporuje pouze parametr `-c` určující počet bajtů k přečtení. Aplikace čte ze standardního vstupu nebo ze souboru zadaného jako argument a zapisuje na standardní výstup.

4.4 Optimalizace

Zde je pár možností, jak redukovat čas procesoru strávený obsluhou DMA kanálů:

1. Jednou z možných optimalizací na straně CPU je provádět přenos data z více bufferů naráz (nejlépe ze všech připravených bufferů). Pro takový přenos dat je nezbytné podporovat timeout, aby bylo možné přenos přerušit v případě vyčerpání dostupných dat z řadiče FPGA.

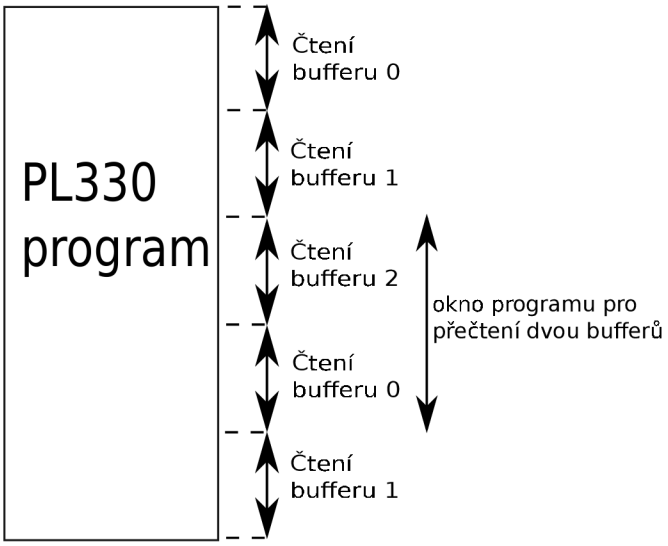
2. Jak bylo popsáno v sekci 4.2, pro každý přenos dat z bufferu je znovu sestavován PL330 program. Speciálně pro čtení z řadiče FPGA, kdy je přenášen vždy stejný počet dat a délek rámců (liší se pouze cílová adresa) lze ušetřit čas tak, že bude program pro PL330 sestaven podle šablony. V takto sestaveném programu stačí před každým přenosem změnit pouze cílovou adresu.

Obě optimalizace mohou být nasazeny současně. Může tak být sestaveno několik podprogramů (každý pro přenos dat z jednoho bufferu). Každý podprogram musí fungovat i samostatně, a tak v každém podprogramu za instrukcemi pro přenos dat musí být instrukce `DMASEV` (vyvolání přerušení) a `DMAEND` (ukončení vlákna). Pokud bude třeba provést více podprogramů v jednom požadavku, lze pro všechny podprogramy, kromě posledního, přepsat instruckce `DMASEV` a `DMAEND` instrukcí `DMANOP`. Jelikož jsou buffery v kruhovém seznamu,

je nutné mít všechny podprogramy pro buffery (kromě posledního) uloženy dvakrát – poté stačí vybrat okno programu, pro jaké buffery má být přenos proveden. Nevýhodou jsou nároky na velikost programu PL330. Počet podprogramů pro přenos jednotlivých bufferů je vyjádřen následujícím vztahem:

$$pocet\_programu = 2 * pocet\_bufferu - 1$$

Obrázek 4.3 zobrazuje situaci s popsanými optimalizacemi, kdy má znakové zařízení dostupné 3 buffery a je potřeba přenést data z bufferu číslo 2 a 0.



Obrázek 4.3: Okno programu pro PL330

Kapitola 5

Testování a závěr

V této práci byla navržena a otestována komunikace mezi user space aplikací a FPGA řadičem pomocí jednotky PL330. Byly vyvinuty dva ovladače – jeden pro obsluhu DMA kanálů jednotky PL330 a druhý pro vytvoření znakových zařízení jako rozhraní pro komunikaci mezi user space aplikacemi a FPGA řadičem. Navržený protokol podporuje přenos rámců (konce rámců lze detekovat i signalizovat).

Testování navržené komunikace proběhlo platformě na Altera SoC, implementované ovladače byly otestovány na Linuxu verze 3.13. Detekce a signalizace rámců byla experimentálně ověřena.

V současné implementaci nejsou ošetřeny chyby vyvolány FPGA řadičem na AXI sběrnici. Pro zotavení se z chyb bude potřeba vyvinout pomocný buffer v rámci FPGA řadiče, který bude uchovávat poslední přenášený burst.

Ovladač znakových zařízení bude vhodné v budoucnu optimalizovat (návrhy optimalizace popsány v sekci 4.4) – současná implementace ovladače znakových zařízení před každým přenosem dat znovu sestavuje program pro jednotku PL330, což je zbytečná režie navíc.

Bylo by zajímavé implementovat fallback ovladač pro případ, že nebude dostatek DMA kanálů pro všechny akcelerátory. Takový ovladač by měl místo DMA přenosů provádět zápis a čtení dat pomocí CPU.

Do budoucna je plánováno přenést ovladače na platformu RSoC Framework.

Literatura

[1] ARM Limited: *AMBA AXI Protocol*. 2004.

[2] ARM Limited: *PrimeCell DMA Controller (PL330)*. 2007.

[3] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux Device Drivers*. O'Reilly Media, Inc, January 2005.

[4] Kotásek, Z.: Periferní zařízení: studijní opora [online]. `http://www.fit.vutbr.cz/study/courses/IPZ/public/.cs`, 2007.

[5] Leemhuis, T.: What's new in Linux 3.10 [online]. `http://www.h-online.com/open/features/What-s-new-in-Linux-3-10-1902270.html`, July 2013.

[6] Masters, J.; Blum, R.: *Linux Profesionálně*. ZONER software, 2008.

[7] Viktorin, J.: *HW/SW Co-design for the Xilinx Zynq Platform*. diplomová práce, FIT VUT v Brně, 2013.

[8] Waugh, T.: kernel-doc nano-HOWTO [online]. `https://www.kernel.org/doc/Documentation/kernel-doc-nano-HOWTO.txt`.

[9] Xilinx, Inc.: *Zynq-7000 All Programmable SoC Overview*. September 2014.

Příloha A

Obsah CD

CD obsahuje následující soubory a adresáře:

- `config/` - Adresář s konfiguračními soubory pro Buildroot a Linux a konfigurací pro DeviceTree.
- `doc/` - Adresář se zdrojovými kódy dokumentace této práce v jazyce LaTeX.
- `src/` - Adresář se zdrojovými kódy bakalářské práce.
- `bp_havran.pdf` - Tato dokumentace v elektronické podobě.

Příloha B

Manual

Pro zprovoznění je doporučen hardware Xilinx Zynq a Altera SoC FPGA a Linux verze 3.13 (testovaná verze). Je nutné zakázat původní Linuxový ovladač pro PL330 (lze použít přiložený konfigurační soubor `linux.config`). Dále je nutné ověřit, aby bylo správně nastaveno periph ID v .dts souboru (lze použít přiložený `zynq-zed.dts`), jinak by nebyl správně identifikován příslušný ovladač pro obvod PL330. Pro použití Linuxu s Buildrootem je přiložen konfigurační soubor (`buildroot.config`).

Pro funkčnost celého systému je nezbytné nahrání modulu `pl330.ko` před modulem `chrdev.ko`.