VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

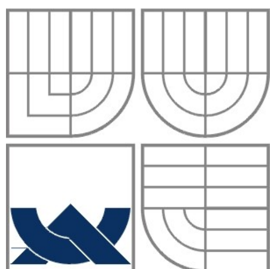
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HAŠOVACÍ ALGORITMY V FPGA

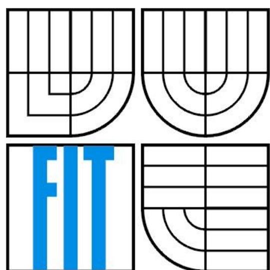
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Jiří Novotňák
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HAŠOVACÍ ALGORITMY V FPGA

HASH ALGORITHMS FOR FPGA

BAKALÁŘSKÁ PRÁCE

BACHELOR´S THESIS

AUTOR PRÁCE Jiří Novotňák
AUTHOR

VEDOUCÍ PRÁCE Ing. Václav Šimek
SUPERVISOR

BRNO 2008

Zadání bakalářské práce

Řešitel: **Novotňák Jiří**

Obor: Informační technologie

Téma: **Hašovací algoritmy v FPGA**

Kategorie: Bezpečnost.

Vedoucí: **Šimek Václav, Ing.**

Pokyny:

1. Seznamte se s oblastí bezpečnostních algoritmů pro zajištění autentičnosti zprávy.
2. Věnujte pozornost hašovacím algoritmům.
3. Prostudujte možnost jejich HW implementace v rekonfigurovatelných obvodech FPGA s pomocí jazyka VHDL.
4. Algoritmus dle vlastního výběru, např. MD5 se pokuste implementovat v FPGA na platformě FITkit.
5. Demonstrujte funkčnost na zvoleném souboru dat.
6. Diskutujte možnosti dalšího vylepšení.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Tato práce se zabývá implementací hašovacího algoritmu MD5 v programovatelném hradlovém poli FPGA umístěném na výukové platformě FITkit. Výsledný celek se skládá ze 2 entit - první provádí rychlý výpočet algoritmem MD5 a druhá obstarává komunikaci mezi FITkitem a počítačem přes rozhraní RS232 pomocí protokolu podle vzoru SLIP. Pro demonstraci funkce je přiložena jednoduchá aplikace, přenášející soubor po zadaném rozhraní do FITkitu a vypisující výsledný haš do aplikačního okna na počítači. Na závěr jsou diskutovány možnosti dalšího rozšíření.

Klíčová slova

FITkit, FPGA, MD5, VHDL

Abstract

This work is dealing with the implementation of MD5 hash algorithm in programmable FPGA circuit on FITkit education platform. Practical results of our effort primarily include two entities - one of them is responsible for a quick computation of MD5 algorithm while the other one handles the communication tasks between FITkit and host enviroment via RS232 link and SLIP-like protocol. The demonstration of the capabilities consists of a simple application, which transfers the given file to FITkit and displays the result of MD5 computation in the application window on PC. Brief evaluation and number of ideas for future improvements are given in the conclusion.

Keywords

FITkit, FPGA, MD5, VHDL

Citace

Jiří Novotňák: Hašovací algoritmy v FPGA, bakalářská práce, Brno, FIT VUT v Brně, 2008

Hašovací algoritmy v FPGA

Prohlášení

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

……………………
Jiří Novotňák
14.května 2008

Poděkování

Tímto děkuji za pomoc a odborné rady vedoucímu projektu Ing. Václavovi Šimkovi.

© Jiří Novotňák, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah

Obsah.....1
1 Úvod.....3
2 FPGA.....4
2.1 Historie technologie FPGA.....4
2.2 Princip obvodu FPGA.....4
2.3 Jazyk VHDL.....5
3 Hašovací algoritmy.....6
3.1 Jednoduché hašovací algoritmy.....6
3.2 Moderní hašovací algoritmy.....6
3.2.1 Požadavky na vlastnosti hašovacích algoritmů.....7
3.2.2 Náhodné orákulum.....7
3.2.3 Princip moderních hašovacích funkcí.....8
3.3 Algoritmus MD5.....8
3.3.1 Princip algoritmu MD5.....8
3.3.2 Vlastnosti algoritmu MD5.....9
4 Implementace algoritmu MD5.....11
4.1 Použité vybavení.....11
4.1.1 Výuková platforma FITkit.....11
4.1.2 FPGA na nástroji FITkit.....12
4.1.3 USB-UART převodník.....12
4.1.4 Překladový systém FITkitu.....12
4.1.5 Programovací prostředí Delphi 6.....12
4.1.6 Komponenta ComPort.....13
4.2 Využití kapacity FPGA čipu.....13
4.3 Struktura návrhu.....13
4.3.1 Zvolená top level entita.....13
4.3.2 Komunikace s počítačem.....14
4.3.3 Struktura top level entity.....15
4.3.4 Entita MD5.....16
4.3.5 Struktura kódu popisující entitu MD5.....17
4.4 Formát přenášených dat.....18

4.4.1 Komunikační protokol....18
4.4.2 Formát výstupních dat....19
4.5 Implementace top level entity....20
4.6 Implementace entity md5....20
4.6.1 Jednotka CONTROLER....21
4.6.2 Jednotka READ....23
4.6.3 Jednotka CORE....24
4.6.4 BlockRAM....25
4.6.5 Časování entity MD5....25
4.7 Konečné nastavení a vlastnosti....26
4.8 Propustnost entity MD5....26
5 Demonstrace funkce a výsledků....28
5.1 Přiložená ukázková aplikace....28
5.2 Použití programu Hyperterminal....28
5.3 Otisky získané z vybraných vstupů....29
6 Možnosti dalšího rozšíření....30
6.1.1 Převod zápisu do strukturální podoby a optimalizace....30
6.1.2 Jiná koncová aplikace....30
6.1.3 Jiné rozhraní....30
7 Závěr....31
Literatura....32
Seznam příloh....33

1 Úvod

Hašovací algoritmy jsou velmi používané funkce ve světě sítí a komunikací, téměř kdykoli je třeba počítat s jejich zabezpečením. I z tohoto důvodu již mají za sebou dlouhý vývoj a existuje velmi mnoho takových algoritmů s různou úrovní bezpečnosti. Většina z nich ale spadá do kategorie kontrolních součtů a jednoduchých algoritmů na pouhou detekci chyby. Bezpečných funkcí, na které se můžeme spolehnout i při extrémně důležitých situacích je relativně málo, navíc čím dál tím víc jich zůstává prolomených.

Mezi nejznámější tohoto druhu patří algoritmus MD5, který se donedávna používal ve většině odvětví informačních technologií týkajících se zabezpečení. Samotný algoritmus je dnes silně prolomený, ale stále se využívá a jeho vlastnosti jsou v mnoha ohledech uspokojivé. Zejména při vytváření kontrolních součtů je mnohem účinnější než dříve používané algoritmy jako checksum, ale třeba i CRC. Toto se může velmi výrazně projevit v případě, kdy je přenášen velký objem dat a pravděpodobnost neobjevené chyby kontrolním součtem je již nepřiměřeně vysoká. V takových situacích algoritmy jako MD5 neztrácí svoji pozici spolehlivého detektoru chyb.

Díky tomuto může být velmi vhodné implementovat takovéto algoritmy v hardwarových zařízeních, které dokáží zpracovat velmi velký průtok dat. Naneštěstí dostupných implementací není mnoho a i přes skutečnost, že se již zřídka objevují i v publikacích, často je o implementaci v čipu o velké kapacitě, kdy je možná zbytečně plýtváno zdroji.

Cílem této práce tedy je implementovat algoritmus MD5 v programovatelném hradlovém poli FPGA [2] umístěném na výukovém nástroji FITkit. Toto hradlové pole je svoji kapacitou velmi omezené, ale nástroje FITkit jsou přístupné všem studentům FIT VUT v Brně a tato implementace by mohla velmi usnadnit pochopení tvorby takových algoritmů mnoha studentům a i v případě praktického nasazení by mohla mít dostatečné vlastnosti na méně náročné použití.

Mojí snahou tedy bylo vytvořit v jazyce VHDL pokud možno kompaktní design realizující algoritmus MD5 provozuschopný na výukové platformě FITkit a mající co možná nejlepší vlastnosti i přes omezení dané nízkou kapacitu použitého hradlového pole.

2 FPGA

FPGA je zkratka z anglického názvu Field-Programmable Gate Array neboli programovatelné hradlové pole. Jedná se o tzv. rekonfigurovatelný obvod, též programovatelný logický obvod, který nemá svoji funkci danou výrobcem, ale může být programován (rekonfigurován) během své činnosti. Takovéto obvody jsou vyvíjeny již několik desetiletí, neboť možnosti jejich využití mohou být velmi velké. Na obvodech FPGA probíhá vývoj logických obvodů, ale i v běžných podmínkách najde své využití. Zpracování například multimediálních informací hardwarovým obvodem je i o několik řádů rychlejší, než stejná činnost prováděná hlavním procesorem osobního počítače. Pro nejběžnější operace se do běžných počítačů přidávají specializované obvody, akcelerující zpracování například 3D scény,. Přestože by to však bylo z hlediska výkonu velmi dobré, není možné integrovat speciální čip pro každé odvětví výpočtů. V takovém případě je však elegantním řešením jeden obvod, který dokáže svou činnost měnit podle aktuálních požadavků.

2.1 Historie technologie FPGA

Technologie rekonfigurovatelných obvodů se vyvíjí od 80. let 20. století. První takové obvody nesly označení PAL (Programable Array Logic) skládající se z pevné matice logických hradel, které se propojovaly programovatelnými jednotkami. Obvody této technologie byly svou kapacitou velmi omezené a dovolovaly implementovat například malý stavový automat. Pro zvýšení kapacity obvodu se začala používat technologie CPLD (Complex Programable Logic Device), jenž se skládala z několika jednotek PAL, které se opět dynamicky propojovaly. Dalším vývojovým stupněm bylo FPGA, kdy se upustilo od pevné matice logických hradel, které byly nahrazeny jednotkami CLB.

2.2 Princip obvodu FPGA

Obvod této technologie se skládá z matice logických bloků označovaných zkratkou CLB (Complex Logic Block). Každý logický blok obsahuje tabulku LUT (angl. Look Up Table), malou paměť, pomocí které se realizuje logická funkce. Současné obvody FPGA obsahují řádově stovky až tisíce bloků CLB, Pro zvýšení efektivity programovatelných hradlových polí jsou součástí matice logických bloků i další prvky, jako paměť RAM, jednotky DSP a implementované násobičky. Celý obvod dále obsahuje vstupně výstupní bloky, které slouží ke komunikaci architektury nahrané do čipu FPGA s okolím. Jednotlivé bloky jsou propojovány podle konfigurace nahrané do statické paměti RAM, stejně jako vlastní LUT tabulky. Z principu konfigurace volatilními pamětmi tedy vyplývá nutnost rekonfigurovat logický obvod při každém připojení elektrického proudu.

2.3 Jazyk VHDL

VHDL je jazyk pro popis chování logických obvodů. Jeho název vznikl zkratkou VHSIC Hardware Definition Language, kde úvodní část je opět zkratkou z Very High Speed Integrated circuits. Počátky tohoto jazyka sahají do začátku 80. let, kdy jeho vznik iniciovalo ministerstvo obrany Spojených Států. Jazyk VHDL se dočkal jako první z jazyků pro popis hardware standardizace IEEE, konkrétně IEEE 1076 a později byl přidán standard IEEE 1164, který definuje hodnoty logických stavů. Jazyk VHDL je nezávislý na cílové platformě. Ovšem díky skutečnosti, že je možné jím popisovat chování logického obvodu velmi abstraktně, není možné provést syntézu (převést popis chování obvodu do skutečných logických jednotek) každé konstrukce zapsané tímto jazykem. Přestože jazyk jako takový je jednoznačný a simulace v něm popsáného obvodu pracuje podle očekávání, vlastní syntéza se nemusí zdařit. Syntetizátor velmi často vrátí chybu, nebo vygeneruje obvod, který se nechová očekávaným a požadovaným způsobem. Navíc různé nástroje pro tuto činnost mohou být různě úspěšné, jelikož vlastní syntéza je velmi komplikovaný proces. Práci syntetizátoru lze však velmi usnadnit (nebo naopak ztížit) díky různým způsobům zápisu kódu ve VHDL. Stejný obvod můžeme popsat jeho chováním, jedná se tedy o behaviorální popis, nebo jeho strukturou, kdy mluvíme o strukturálním popisu. Zatímco v prvním případě musí syntetizátor takřka vymyslet obvod, strukturální popis převede velmi snadno. Je tedy již při psaní kódu v jazyce VHDL možné velmi výrazně ovlivnit úspěšnost, popřípadě i přenositelnost tohoto kódu. Tyto vlastnosti se však mohou dobře využít při návrhu složitějších obvodů. Lze například navrhnout behaviorální popis architektury, simulací odzkoušet její vlastnosti a posléze postupně přepisovat kód do strukturální podoby, nebo alespoň jednodušších behaviorálních celků. Navíc není nutné složitě popisovat obvod, který nemáme v úmyslu syntetizovat, jako například různé testovací entity.

3 Hašovací algoritmy

Pod pojmem hašovací (angl. hash) rozumíme algoritmus, který ze vstupu o předem nedefinované délce vrátí tzv. otisk, neboli řetězec o pevně dané délce. Podmínkou je, aby ze stejného vstupu algoritmus vygeneroval stejný výstup a při jakékoli změně vstupu byl výsledný otisk jiný.

3.1 Jednoduché hašovací algoritmy

Nejjednoduššími hašovacími funkcemi jsou takzvané kontrolní součty (angl. checksum), které sčítají ASCII hodnoty znaků na vstupu a vrací číslo o pevné bitové délce dané jejich součtem. Dalším stupněm jsou algoritmy založené na logické funkci xor, například známy CRC. Tyto algoritmy slouží většinou ke stejnému účelu a to pro ověření, zda při přenosu nedošlo k poškození dat, rozdíl je pouze ve schopnosti detekovat případné změny. První zmiňovaný kontrolní součet je relativně málo účinný. Je logické, že jelikož se znaky pouze sčítají, mají například různé permutace jednoho řetězce stejný součet, přestože se rozhodně nejedná o stejné řetězce. Algoritmus CRC je na tom o poznání lépe, jeho účinnost se odhaduje na více než 99 procent. Z tohoto důvodu se zejména on uchytil jako kontrolní algoritmus při přenášení dat, či kontrole integrity dat při použití komprimace a v mnoha dalších oblastech. I přes tyto vlastnosti je však příliš slabý, pokud chceme mít 100% jistotu, že se daný soubor dat nezměnil. Jeho slabosti se projeví zejména ve chvíli, kdy existuje útočník, který se snaží tuto kontrolu cíleně obejít. S jistou dávkou námahy je totiž možná přidat k souboru data tak, aby CRC kód zůstal nezměněn a tím například vnést do spustitelného souboru virus, nebo jinak změnit jeho činnost. V dnešní době, kdy se přes celosvětovou síť internet mohou provádět vysoké finanční transakce, kdy jsou přes tuto síť přenášeny často velmi citlivé údaje a kdy počet uživatelů této sítě prudce roste a s ním i počet potenciálních i skutečných útočníků, nestačí funkce dokazující, že na asi 99% je daný soubor ten, za který jsme chtěli, nebo že digitální podpis zřejmě patří osobě, která ho předkládá atd. V momentě, kdy se někdo přihlašuje ke svému bankovnímu účtu, popř. provádí finanční transakci, je vyžadována 100% důvěryhodnost. Minimální možnost rozšíření stávajících funkcí by mohlo být zvýšení počtu bitů výsledných otisků a tím ztížení hledání kolizních řetězců. Přesto by však ze svého relativně jednoduchého principu nebyly schopny dosáhnout dostatečně vysoké bezpečnostní úrovně.

3.2 Moderní hašovací algoritmy

Základní požadavky na hašovací algoritmus jsou zřejmé, když se však nad tímto problémem zamyslíme hlouběji, zjistíme, že je zajistit 100% jistotu, je nedosažitelné. Máme-li například 1MiB

soubor, z něhož vygenerujeme řekněme 128 bitový otisk, navíc takový, ze kterého není možné zjistit původní zprávu, je zřejmé, že došlo ke ztrátě informací, a tedy v případě vhodné souhry okolností musí existovat další řetězce, které generují stejný výsledný otisk. Dosažení nemožnosti existence kolidujících řetězců je tedy samo o sobě nemožné. V praxi se tedy musíme spokojit s vědomím, že prolomení možné je, ale s výpočetní technikou současnosti a předpokládané blízké budoucnosti by prolomení trvalo příliš dlouho (třeba i stovky let). Hašovací funkce tedy musí být postavena tak, aby nalezení kolidujících řetězců bylo možno jen tzv. hrubou silou, tedy lidově řečeno metodou pokus-omyl, a aby tato cesta nebyla reálně použitelná. Takovýto postup totiž silně komplikuje počet možných kombinací otisků, jenž je daný jejich délkou. Toto je na první pohled asi nejvýraznější rozdíl, když vidíme výstup například CRC s 32 bity a například SHA-1 se svými 160 bity. Další požadavky popíši v následující kapitole.

3.2.1 Požadavky na vlastnosti hašovacích algoritmů

Mít funkci vracející dlouhý otisk je účinné, pokud budeme předpokládat, že útočník se nepokusí najít kolizní řetězce jinak než hrubou silou. Bohužel však rozborem dané funkce je většinou možno snížit celkový počet kombinací jejích výstupů a tím pádem při nejmenším velmi zrychlit útok. Proto se každá hašovací funkce podrobuje důkladnému zkoumání a má-li být považována za bezpečnou je nutno, aby nebylo možné v přijatelném čase nalézt:

1. Vstupní soubor, podle znalosti jeho otisku
2. Podle již existujícího vstupního souboru najít další, který by funkcí generoval stejný výstup.
3. Jakékoli dva rozdílné vstupy generující stejný výstup

První 2 podmínky u dnešních hašovacích funkcí většinou bývají dodrženy, problematický je ale bod 3. Je to způsobeno faktem, že se generují 2 libovolné vstupy a je tedy mnohem snažší najít kolidující řetězce.

3.2.2 Náhodné orákulum.

Jako ideál hašovacích funkcích je v teorii považováno tzv. náhodné orákulum (viz [4]). Jde o teoretické zařízení, které z jakéhokoli vstupního souboru dat vygeneruje zcela náhodně výstupní řetězec. Pokud však jednou nějaký vstupní soubor zpracuje, již si pamatuje výstupní kód a při dalším zpracovaní jej vrátí místo náhodného výsledku. Není tedy možné deterministickou cestou porušit kteroukoli ze 3 podmínek uvedený výše. Prakticky však není v našich silách vymyslet zařízení, které by se chovalo takovýmto způsobem, které by bylo náhodné a zároveň deterministické tak, jak bychom potřebovali.

3.2.3 Princip moderních hašovacích funkcí

Vývoj hašovacích funkcí dospěl v posledních letech do bodu, kdy se jejich princip velmi podobá. Obecně lze říci, že funkce mají pevný řetězec, většinou předem smluvený, který postupně šifrují klíčem, kterým se stávají vstupní data. Šifrování probíhá postupným aplikováním logických funkcí a sčítáním, či aplikace funkce xor. Tento postup probíhá opakovaně, tzv. iteračně, a jednotlivé funkce aplikované při šifrování se střídají a tvoří tzv. kaskádu, kdy je jedna funkce aplikovaná na výstup funkce předchozí. Tento postup je z dnešního pohledu relativně bezpečný, ale neposkytuje takový stupeň ochrany, jak by se mohlo zdát. Přesto je však využit jak u MD5 nebo algoritmu SHA-1 u kterého byly také objeveny menší slabiny, ale i u rodiny algoritmů SHA-2 která podle dostupných informací zůstává dodnes jakkoli neprolomena.

3.3 Algoritmus MD5

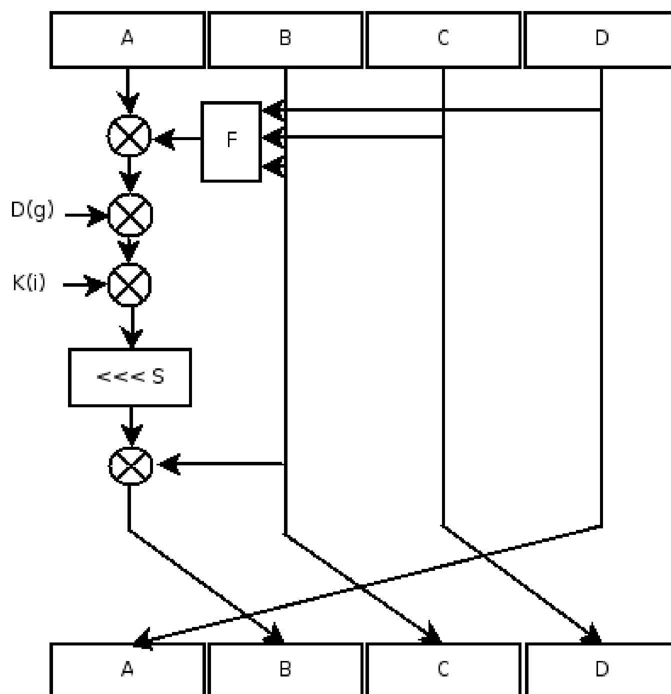
Dnes asi nejznámější hašovací funkcí je MD5. Tento algoritmus byl prezentován roku 1991 Ronaldem Rivestem, jako náhrada staršího a slabšího algoritmu MD4, a dubnu 1992 byl standardizován jako RFC1321 [1]. Funkce MD5 oproti svému předchůdci přenáší vývojové změny, například provádí větší počet iteračních cyklů. Algoritmus MD5 se zejména v 90. letech 20. století velmi používal v různých aplikacích od kontrolního součtu souborů přes šifrování hesel až k ověřování digitálních podpisů. V dnešní době se od této funkce v aplikacích vyžadujících vysoký stupeň zabezpečení upouští a je nahrazována rodinou funkcí SHA.

3.3.1 Princip algoritmu MD5

Algoritmus rozdělí vstupní data na shluky o velikosti 512 bitů. Každý shluk je dále rozdělen na 16 32 bitových částí, které se použijí jako klíč k šifrování 128 bitového vektoru. Ten může mít libovolnou počáteční hodnotu, čímž se ale změní výsledný otisk. Proto v případě potřebné přenositelnosti je podle RFC definován jako $67452301EFCDAB8998BADCFE10325476_{16}$, což jsou 4 čísla 01234567_{16}, $89ABCDEF_{16}$, $FEDCBA98_{16}$ a 76543210_{16} načtené ve formátu little endian. Těchto 128 bitů udržuje kontext mezi zpracovávání jednotlivých shluků dat a po dokončení poslední smyčky se stává výsledným otiskem.

Poslední 512 bitový shluk dat, který zůstává nedokončený se zarovnává. Za poslední bit dat se přidá 1 bit logické 1 a koncových 64 bitů posledního 512 bitového shluku obsahuje bitovou délku vstupních dat. Prostor mezi přidanou 1 a počátkem bitové délky je vyplněn logickými 0. V případě, kdy mezi připojenou logickou 1 a hranicí shluku není dostatek místa, je délka dat umístěna až nakonec nového shluku, kdy je nevyužitý prostor opět vyplněn logickými 0.

Vnitřní tělo algoritmu provádí nad každým shlukem dat celkem 64 iterací rozdělených na 4x16, po kterých se mění logická funkce a permutace vstupních dat. Algoritmus pracuje s 32 bitovými čísly na které dělí jak vstupní data, tak i 128 bitový vektor tvořící kontext mezi jednotlivými iteracemi. Každé 32 bitové číslo, jenž je načteno je chápáno jako číslo ve formátu little endian zcela stejně, jak by jej zpracoval 32bitový procesor x86. Při každé iteraci je nejprve provedena jedna ze 4 logických funkcí nad 3 částmi 128bitového kontextu a výsledek je přičten k 4. (resp. 1.) části, dále je přičtena jedna 32bitová část datového shluku, jedna z 16 konstant *k*, výsledek je rotován podle aktuální konstanty *r*, je přičtena 2. 32bitová část kontextu a výsledek je uložen do 2. 32bitů kontextu bráno zleva. Zbytek kontextu je rotován o 32bitů doprava. Následující obrázek, názorně ilustruje vlastní algoritmus:



Obrázek 3.1: Jedna z 64 iterací prováděných nad každým 512 bitovým blokem dat

3.3.2 Vlastnosti algoritmu MD5

Tento algoritmus působí na první pohled velmi sofistikovaně je v něm patrný evoluční vývoj rodiny funkcí MDx. Však již roku 1993 byly objeveny kolizní řetězce, roku 1996 menší vada návrhu a roku 2004 algoritmus na vyhledávání kolizních řetězců. Jeden z problémů funkce MD5 je 128 bitový výstup. Vzhledem vývoji výpočetní techniky je takto krátký otisk příliš snadno prolomitelný hrubou silou i kdyby algoritmus jiné slabiny neobsahoval. Další kritizovanou vlastností tohoto algoritmu je procházení vstupního souboru pouze jednou. Toto velmi usnadňuje implementaci, ale v okamžiku,

kdy se během zpracovávání 2 rozdílných řetězců podaří útočníkovi dostat do stavu, kdy mají stejný kontext, již mu v nalezení kolize téměř nic nebrání. Nutno však podotknout, že rodina algoritmů SHA, kterou se v současnosti nahrazuje MD5 v aplikacích s důrazem na bezpečnost, tuto vlastnost zachovává a přesto algoritmy řady SHA-2 prozatím nejsou jakkoli prolomeny.

Dalším problémem této funkce je její známost a skutečnost, že její prolamování se pro mnohé stalo jakýmsi koníčkem. Existují internetové servery s databází hesel a jejich otisků, pokud se tedy útočník dostane otisku hesla a dané heslo není dostatečně originální, jeho rozluštění může být otázkou několika vteřin a nepomůže ani dostatečná délka hesla. Někdy je možné dokonce získaný otisk vložit do fulltextového vyhledávače a získat nezašifrovanou podobu hesla.

Algoritmus MD5 se tedy dnes nepovažuje za dostatečně bezpečný a u technologií zaměřených na bezpečnost se od něj upouští. Jeho vlastnosti jsou však v méně náročných aplikacích dostačující, zejména při počítání kontrolních součtům zpráv i souborů.

4 Implementace algoritmu MD5

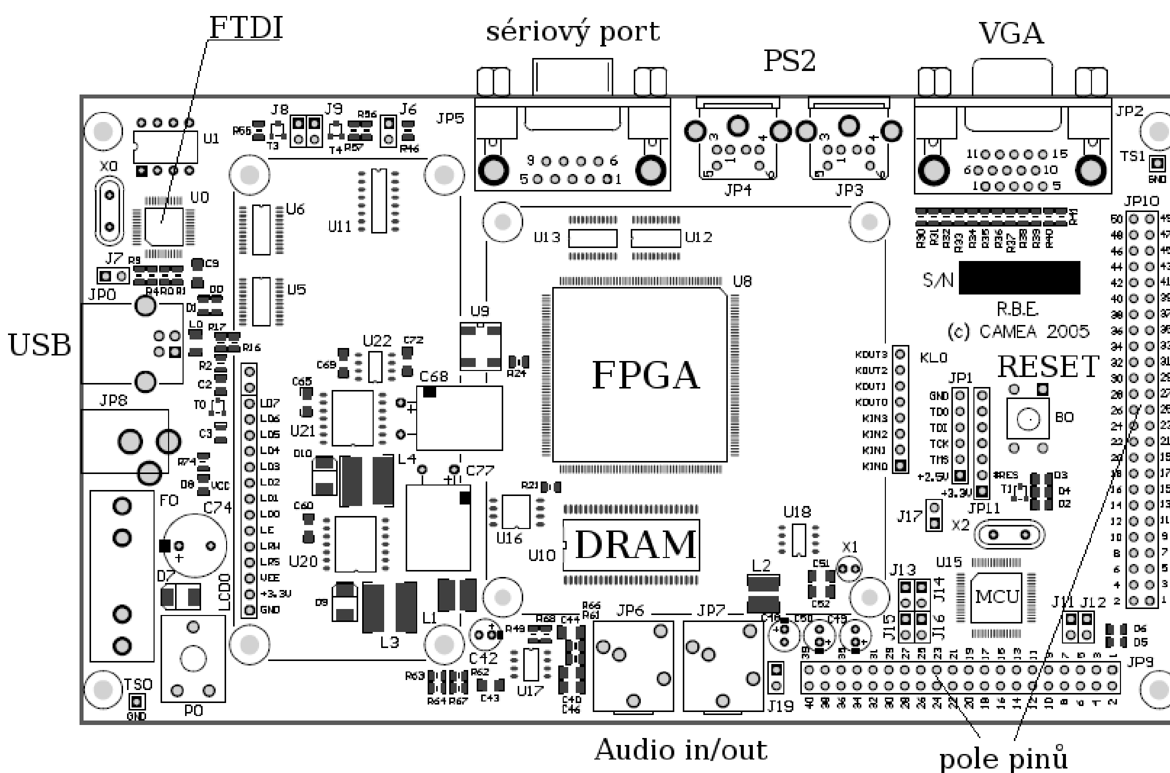
V této kapitole se budu zabývat vlastní implementací hašovacího algoritmu MD5 v programovatelném hradlovém poli FPGA umístěného na výukovém nástroji FITkit.

4.1 Použité vybavení

Již v zadání bakalářské práce byla specifikována cílová platforma. Nástroj FITkit, který je oním cílem je však velmi univerzální a obsahuje mnoho různých zařízení a komunikačních rozhraní. Z tohoto důvodu zde v krátkosti popíši konkrétní technické vybavení, které bylo využito.

4.1.1 Výuková platforma FITkit

Tento nástroj je určen k získání praktických zkušeností s vestavěnými systémy. FITkit obsahuje výkonný mikrokontrolér MSP430F168IPM od firmy Texas Instruments, 8x8MBit paměti DRAM, USB-UART převodník FT2232C umožňující komunikaci 2 nezávisle konfigurovatelnými kanály RS232 nebo RS422/RS485, Audio vstupy/výstupy, konektory PS2 pro připojení myši/klávesnice, konektor VGA, samostatný konektor cannon9 s rozhraním RS232, 16 tlačítkovou klávesnici, jednořádkový LCD display, 2 pole pinů a pro tuto práci především programovatelné hradlové pole



Obrázek 4.1: PCB nástroje FITkit

informace o této platformě viz webové stránky projektu FITkit [2]. Na obrázku 4.1 je znázorněno PCB FITkitu včetně popisů některých důležitých částí.

4.1.2 FPGA na nástroji FITkit

Výuková platforma FITkit, na kterou je tato práce směřována, je opatřena nejmenším FPGA čipem řady Spartan 3 od společnosti Xilinx. Jedná se o obvod XC3S50-4PQ208C [5]. Tento čip obsahuje 192 konfigurovatelných bloků CLB uspořádaných do matice o 16 řádcích a 12 sloupců. Celkem obsahuje 1728 logických buněk a 50000 logických hradel. Dále jsou v tomto čipu přítomny 2 jednotky pro správu hodin, 72kbitů paměti blockRAM rozdělené do 4 bloků, 12kbitů distribuované paměti RAM a 4 násobičky 18x18 bitů. Toto programovatelné hradlové pole komunikuje s mikrokontrolérem pomocí sériové sběrnice SPI. Konfiguraci FPGA je možné provádět za pomoci mikrokontroléru, který je připojen na jeden sériový RS232 kanál USB-UART převodníku. Není tedy nutný speciální kabel JTAG a při vhodném nastavení propojek mikrokontrolér po resetu automaticky nakonfiguruje FPGA z připojené FLASH paměti a je tedy umožněna práce FITkitu nezávislá na připojení k počítači.

4.1.3 USB-UART převodník

K dodávání dat do hradlového pole je využito sériového rozhraní RS232 připojené přes výše zmiňovaný převodník USB-UART a vlastní komunikace probíhá pomocí kabelu USB A-B kabelu připojeného k počítači.

4.1.4 Překladový systém FITkitu

Syntéza kódu byla provedena za použití překladového systému vytvořeného pro platformu FITkit (viz [2]). Jako syntetizační nástroj byl použit program *xst* coby součást nástroje *ISE* verze 9.1 od společnosti Xilinx.

4.1.5 Programovací prostředí Delphi 6

Demonstrační aplikace je vytvořena v programovacím jazyku a prostředí Borland Deplhi verze 6. Jedná se o sofistikované vývojové prostředí postavené na programovacím jazyku pascal. Tato verze je již poněkud starší a nově není podporována. Důvod k jejímu zvolení se stala skutečnost, že je to poslední verze, pro kterou je určena využitá komponenta pro práci se sériovým portem.

4.1.6 Komponenta ComPort

Toto je komponenta programovacího prostředí Delphi do verze 6 od firmy WinSoft. Zajišťuje práci se sériovým portem na velmi dobré úrovni. Jedná se o komerční knihovnu a v ukázkové aplikaci je použita její zkušební verze, která z funkčního hlediska má pouze omezení v nutnosti potvrdit upozorňovací dialog. Zkušební verze byla vybrána, protože i přikládaná aplikace je pouze ukázkou přenosu dat použitým protokolem.

4.2 Využití kapacity FPGA čipu

Při syntéze návrhu jsem došel k poznání, že jsem velmi přecenil možnosti hradlového pole FPGA použitého na FITkitu a byl jsem nucen k jistým úpravám, které se dotkly výkonu výsledného designu a v původní verzi i funkčnosti, kdy jsem byl nucen omezit maximální velikost vstupních dat.

Při konečných úpravách jsem však náhodou odstranil drobnou část kódu, přesněji řečeno jednu podmínku, jež však již neovlivňovala funkčnost návrhu a k mému překvapení syntetizátor poté vygeneroval konfigurační soubor, který byl asi o 50 logických bloků menší. Díky tomuto jsem mohl odstranit funkční omezení a zprovoznit design s původními vlastnostmi.

Výsledný kód však po syntéze opět téměř vyplňuje čip FPGA a při testech se mi několikrát přihodilo, že syntetizátor z neznámých příčin vygeneroval konfigurační soubor, který se opět do čipu nevešel. Tato situace nikdy nenastala se zjednodušenou verzí návrhu, která navíc při použití sériového a tudíž relativně pomalého rozhraní podává stejně dobrý výkon jako plnohodnotná, ale objemnější verze.

Z tohoto důvodu jsem se rozhodl do své práce zahrnout verze obě. Plnohodnotnou, ve které běží jednotlivé procesy paralelně, i zjednodušenou, ve které jsou tyto procesy spouštěny sériově. V textu budu popisovat plnohodnotnou a uvedu implementační rozdíly zjednodušené verze.

4.3 Struktura návrhu

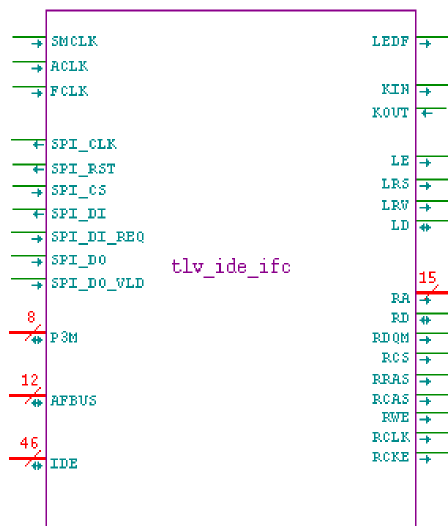
Při návrhu architektury kódu jsem chtěl vytvořit do jisté míry univerzální a přenositelnou entitu, provádějící výpočet, která bude funkčně nezávislá na zvolené platformě. Tato entita měla mít vlastní rozhraní pomocí něhož by měla být připojena k tzv. top level entitě, která bude tvořit vrstvu rozhraní mezi konkrétním komunikačním vybavením dané platformy a již zmiňovanou výpočetní entitou.

4.3.1 Zvolená top level entita

Součástí vývojového prostředí pro programování FPGA na nástroji FITkit jsou na výběr 3 top level entity zpřístupňující komunikační rozhraní FITkitu. Tyto entity se liší zejména využitím sběrnice X

na kterou jsou připojeny jednotlivé komunikační rozhraní. Nejjednodušší je entita *tlv_bare_ifc*, která sběrnici X nepoužívá ani nezpřístupňuje pomocí konkrétních pojmenovaných signálů. Dále je přítomna entita *tlv_pc_ifc*, která má být určena pro aplikace využívající komunikační rozhraní RS232, PS2 a výstup VGA. Poslední top level entita nese označení *tlv_ide_ifc*. Ta je pro implementaci zařízení, která nepoužívají vestavěné konektory a rozhraní. Typickým příkladem, pro které byla tato entita zřejmě i vytvořena je řadič disku IDE, využívající 40 pinové rozhraní. Entita obsahuje port s označením IDE, který zpřístupňuje celou sběrnici X. Díky tomuto přístupu je tato entita velmi variabilní a umožňuje realizaci velmi širokého okruhu zařízení. Například je možné vybrat si některé piny a realizovat přes ně sběrnici jako je I2C, nebo téměř jakoukoli jinou, jíž dostačují parametry, které je FITkit schopen poskytnout a počet dostupných pinů.

Při výběru top level entity jsem měl jediný požadavek, čímž bylo přístupné sériové rozhraní RS232 pokud možno pomocí USB-UART převodníku. Toto rozhraní je připojeno přes sběrnici AFBUS ke které mají přístup entity *tlv_pc_ifc* a *tlv_ide_ifc*. V SVN stromu pro FITkit je ukázková aplikace RS232_echo, která předává znaky mezi terminálem a druhým sériovým rozhraním RS232. Po vzoru této aplikace jsem, jelikož sběrnici X nevyužívám, zvolil jednodušší entitu *tlv_ide_ifc*, jejiž znázorňuje následující obrázek:.



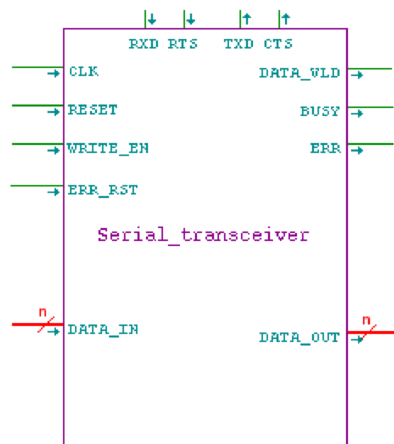
Obrázek 4.2: Entita tlv_ide_ifc

4.3.2 Komunikace s počítačem

Ke komunikaci s počítačem je využito sériové rozhraní RS232 připojené přes USB-UART převodník FTDI 2232 a kabelem USB k počítači. K realizaci signálů rozhraní RS232 je využita entita *serial_transceiver* z SVN stromu FITkitu. Tato entita zajišťuje zpracování signálů rozhraní RS232

a vytváří rozhraní snadno použitelné pro ostatní entity. Krom vlastních řídících signálů obsahuje entita n bitové datové porty pro příjem a vysílání dat. Velmi pěknou vlastností je možnost snadno nastavit parametry spojení, tedy bitovou rychlost a paritu. Celá entita je složena, jak její funkce i název napovídá, ze dvou samostatných entit pro sériovou komunikaci: *serial_transmitter* a *serial_receiver.* Entitu *serial_transceiver* a její rozhraní znázorňuje obrázek 4.3.

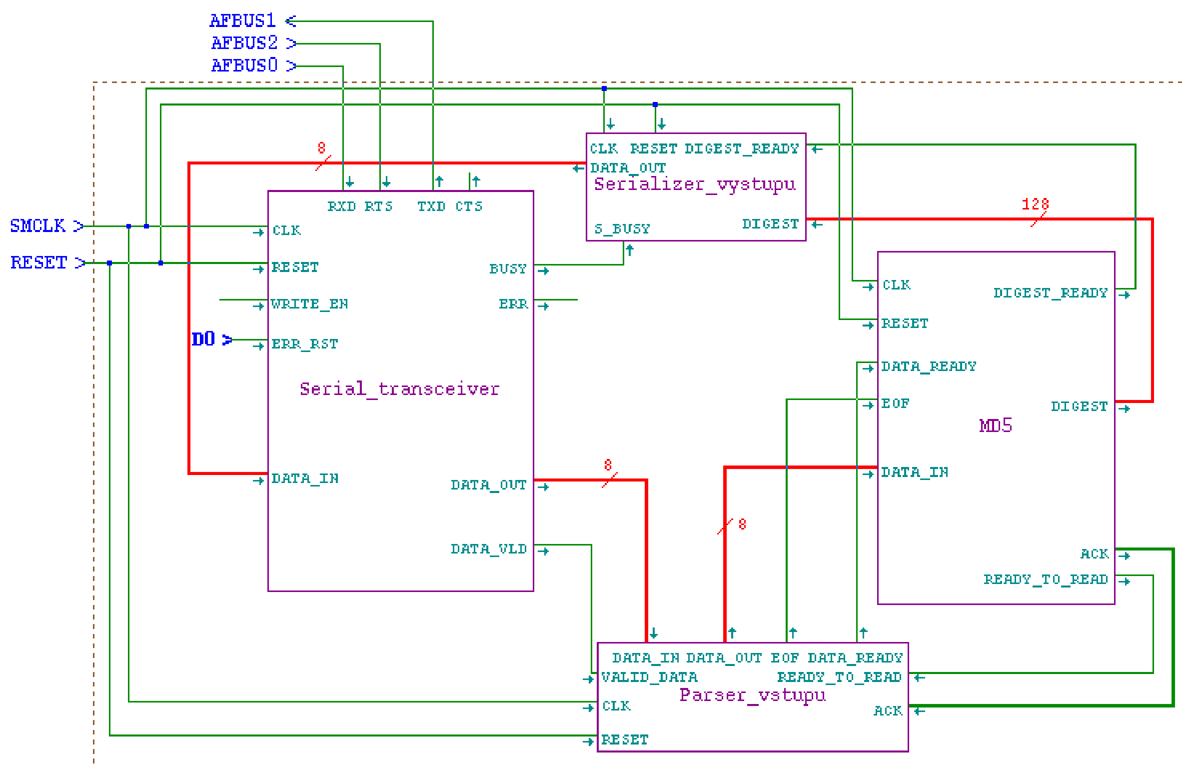
U této entity jsou využity signály *serial_vld* informující o načtení slova požadované délky, *serial_busy* sloužící při odesílání dat pro informaci, zda je linka zaneprázdněna, nebo zda je možné vysílat, *serial_we*, kterým se nastavuje požadavek na odeslání dat rozhraním a signály CLK a RESET. Signál ERR je nepřipojen a ERR_RST nastaven na logickou 0. Chybové stavy přenosového média jsou tedy ignorovány.



Obrázek 4.3: Entita serial_transceiver

4.3.3 Struktura top level entity

Top level entita, jak již bylo zmíněno, souží především jako odstiňující obálka zajišťující propojení entity MD5 se vstupy a výstupy FPGA a též celého zařízení FITkit. Pro úplnost zde tedy uvádím schéma zapojení jednotlivých entit z nichž je je tato „obálka“ sestavena (obrázek 4.4). Mezi entitou serial_transceiver a entitou MD5 jsou procesy zpracovávající vstupní komunikační protokol a serializující výstupní data.

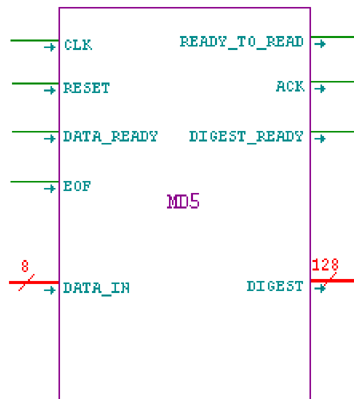


Obrázek 4.4: Struktura top level entity

4.3.4 Entita MD5

Vlastní činnost algoritmu zajišťuje entita se jménem MD5, jejíž strukturu znázorňuje obrázek 4.5. Ta komunikuje s okolím vstupním 8 bitovým portem DATA_IN, kterým se načítají zdrojová data, výstupním 128 bitovým portem DIGEST, na který je přiveden výsledný otisk a signály:

- READY_TO_READ - Výstupní signál aktivní v logické 1, oznamující připravenost k načtení dalších 8 bitů dat.
- ACK - Výstupní signál aktivní v logické 1, oznamující úspěšné přijetí dat
- DIGEST_READY - Výstupní signál aktivní v logické 1, oznamující dokončení práce a aktuálnost otisku na výstupním portu
- DATA_READY - Vstupní signál aktivní v logické 1, informující o připravenosti dat na vstupní sběrnici
- EOF - Vstupní signál aktivní v logické 1, informující entitu o konci vstupních dat.
- RESET - Vstupní signál aktivní při logické 1. Při tomto signálu se entita uvede do stavu po kterém je připravena zahájit od znovu svoji činnost
- CLK - Vstupní signál pro připojení hodinového signálu



Obrázek 4.5: Entita MD5

Za zmínku jistě stojí použití samostatných signálů pro informaci o schopnosti/úspěšnosti čtení. Teoreticky by tyto signály bylo možné nahradit jediným. Důvod použití tohoto schématu je zajištění maximální propustnosti. Každá data přiváděná na sběrnici/port musí být přivedena dřív, než je aktivována entita pro níž je jsou data určena. V případě, kdy je použit pouze jeden hodinový signál, který není nějakým způsobem násoben se většinou data přivádí při opačné hraně hodinového signálu, než je ta, která aktivuje cílovou entitu. V mém případě se entita MD5 aktivuje při vzestupné hraně hodinového signálu a data spolu se vstupními řídícími signály jsou připojovány při hraně sestupné. Informace o neschopnosti číst by se na výstupní signál dostala tedy až při další náběžné hraně hodinového signálu, popř. již při sestupné, ale to ve stejný okamžik, kdy by byla tato informace vyžadována a tedy není zaručen dostatečný předstih nastavení signálů před jejich čtením. Vzhledem k faktu, že načtení 512 bitů 8 bitovou sběrnicí zabere minimálně 64 taktů, jeho zpracování trvá 66 taktů (úvodní načtení, 64 iterací a přičtení výsledku ke kontextu). Přestože konečná implementace tuto vlastnost bohužel upouští původní a logický návrh tyto operace nechává provádět souběžně. V případě nějaké optimalizace, nebo třeba 1 taktového pozastavení vstupu se dostáváme na rozdíl jediného taktu. S touto koncepcí je nadřazená entita schopna rozeznat, jestli se nepodařilo načíst jen jeden blok dat, či jestli se jedná o dlouhodobější indispozici. Díky tomuto je možné maximálně využít všech taktů hodinového signálu. Navíc vzhledem k faktu, že signál READY_TO_READ je též nastavován při sestupné hraně hodinového signálu, jak popíšu později, nelze se pouze na něj v některých případech spoléhat. Posunutí jeho časování by přidalo pouze další latence a problémy.

4.3.5 Struktura kódu popisující entitu MD5

Celá entita MD5 je tvořena jedním souborem obsahujícím jednotlivé procesy tvořící celou entitu. Důvodem pro použití tohoto schématu byla úspora logických bloků. Původní návrh byl členěn

na jednotlivé sub-entity, jež byly samostatně testovány. Při sestavení však nevystačily logické bloky na celý design a jedno z úsporných opatření bylo spojení entit do jedné, což do jisté míry úsporu přineslo.

4.4 Formát přenášených dat

Data přijímaná z počítače, či jiného zařízení na vstupním portu mohou být libovolně dlouhá, teoreticky i nekonečná, jelikož přetečení jejich délky nevede k žádné nepředvídatelné činnosti. Na druhou stranu je třeba vědět, kdy data končí, jelikož se ke zpracovaným bitům připojuje jejich počet. Nelze tedy například průběžně posílat aktuální kontext a když data přestanou přicházet, ten by se stal výsledkem. Je tedy nutné využít přenosový protokol, který by zajistil přinejmenším ukončení přenosu.

4.4.1 Komunikační protokol

Při zamýšlení nad tímto problémem jsem se rozhodoval mezi 3 variantami. První bylo použít existující protokol. V úvahu přicházel například kermit, nebo protokol Xmodem. Složitější protokoly jako Zmodem by byly zbytečně sofistikované, jejich vlastnosti by byly stěží využity, pouze by zesložiťovaly celý návrh, zabíraly zbytečně přenosové pásmo a co by se v konečném důsledku projevilo zřejmě jako fatální by bylo složité dekódování, které by již samo o sobě zabralo příliš mnoho bloků již tak dost využitého hradlového pole s nepříliš velkou kapacitou.

Ze dvou zbylých protokolů jsem si pro další fázi vybral Xmodem. Použití tohoto protokolu by mělo velké výhody. Jednou z nich je zpětná vazba, díky které by v případě rychlé linky, kdyby nakonfigurované hradlové pole nestačilo zpracovávat příchozí data by bylo možné, nebo spíše velmi snadné přenos dat zpomalit, nebo spíše pozastavit, a nedošlo by k poškození vstupních dat. Druhá neméně důležitá výhoda je široká podpora tohoto protokolu. Součástí většiny operačních systémů jsou terminály schopné přenášet soubory tímto protokolem, popřípadě takovéto programy lze zdarma sehnat a snadno doinstalovat. Stačilo by tedy přijmout soubor a vrátit ASCII řetězec výstupného haše a vše by fungovalo velmi pěkně a průzračně.

To vše by bylo však hezké pouze v případě, kdy by bylo v FPGA čipu na FITkitu dostatek nevyužitých logických bloků pro implementaci dekodéru takového protokolu. V opačném případě by to ale znamenalo velmi závažný problém a byla by při nejmenším velká škoda, kdyby výsledný design nebyl použitelný z důvodu příliš komplexního dekódování vstupního protokolu.

Po zvážení výhod a nevýhod jsem se vrátil k variantě, nad níž jsem uvažoval úplně nejdříve. Jednalo se o implementaci protokolu SLIP, který je určen pro přenos IP rámců po sériových linkách. Tento protokol se od předchozích uvedených, také kvůli zcela jinému zaměření, velmi liší.

Neobsahuje žádnou zpětnou vazbu, prakticky je pouze schopen přenést jakákoli data a označit jejich konec. Díky této jednoduchosti je ale režie potřebná na přenos velmi nízká a zejména dekódování je velmi jednoduché. Nevýhodou je, že celkem logicky asi neexistuje program, který by přenášel soubor tímto protokolem. Jeho jednoduchost však byla pro moji implementaci klíčová a rozhodl jsem se tedy data posílat touto cestou. Tento protokol pracuje na jednoduchém principu, kdy za konec přenášených dat vloží speciální znak ($C0_{16}$), jenž značí konec jednoho rámce. V případě výskytu tohoto znaku v přenášených datech je nahrazen sekvencí DB_{16} DC_{16} a v případě výskutu znaku DB_{16} je ten nahrazen sekvencí DB_{16} DD_{16}.

Při implementaci jsem ale došel k závěru, že i tento protokol je pro mé účely příliš náročný na dekódování. Nakonec jsem tedy šel cestou návrhu protokolu vlastního. Při tomto jsem vyšel ze vzoru zmiňovaného protokolu SLIP, ale zjednodušil jsem používání ESC sekvencí, které bylo podle mého názoru zbytečně složité. Výsledný protokol označuje konec přenášených dat znakem 04_{16}, jenž je znakem EOT (EOF) v tabulce ASCII. V případě výskytu tohoto znaku ve vstupních datech je vytvořena ESC sekvence, kdy před problémový znak je vložen znak $1B_{16}$ (ESC v ASCII tabulce), stejně se postupuje při výskytu znaku $1B_{16}$ v datech. Teoreticky platí tedy pravidlo, že znak následující těsně po znaku $1B_{16}$ je brán čistě jako datový a nemá žádnou sémantickou hodnotu. Pro názornou ukázku činnosti protokolu uvádím příklad přenášených dat:

Příklad binárních dat majících být přeneseny:

41_{16} 42_{16} 43_{16} 44_{16} 45_{16} 46_{16} 47_{16} $\mathbf{1B}_{16}$ 48_{16} 49_{16} 50_{16} $\mathbf{04}_{16}$ 51_{16} 52_{16} 53_{16} 54_{16} 55_{16} 56_{16} 57_{16}

Data budou při vlastním přenosu zakódována do podoby:

41_{16} 42_{16} 43_{16} 44_{16} 45_{16} 46_{16} 47_{16} $\mathbf{1B}_{16}$ $\mathbf{1B}_{16}$ 48_{16} 49_{16} 50_{16} $\mathbf{1B}_{16}$ $\mathbf{04}_{16}$ 51_{16} 52_{16} 53_{16} 54_{16} 55_{16} 56_{16} 57_{16} $\mathbf{04}_{16}$

4.4.2 Formát výstupních dat

Výstupní data mají pevnou délku, jsou tedy přenášena bez jakékoli přidané režie. Přesto je však možné použít 2 způsoby jejich formátu. Prvním je formát binární, kdy je postupně přeneseno všech 128 bitů po 8 bitech. Tento přístup umožňuje aplikaci na straně počítače se získaným otiskem naložit podle svého uvážení a je využit i při použití přiložené ukázkové aplikace. Druhým způsobem je použití překladu na ASCII znaky. V tomto případě se přenáší ASCII znaky hexadecimálních čísel jednotlivých čtveřic bitů tak, jak se většinou otisky hašovacích funkcí prezentují. Přenášených dat je v tomto případě dvojnásobek (z každých 4 bitů je vytvořen 8bitový znak). Tohoto lze využít při použití některého terminálu pro komunikaci po sériové lince (například programu hyperterminal) a není třeba dekódovací aplikace (problém nastává samozřejmě se vstupními daty, které musí odpovídat přenášecímu protokolu). Výběr formy výstupu se provádí zakomentování/odkomentování připravených částí v top level entitě a následnou syntézou.

4.5 Implementace top level entity

Entita tlv_ide_ifc se skládá ze 3 procesů. První se aktivuje při vzestupné hraně signálu serial_vld. Tento proces při své aktivaci přijme data ze sériového rozhraní, uloží je do vstupní fronty a zvýší pozici posledních dat ve frontě.

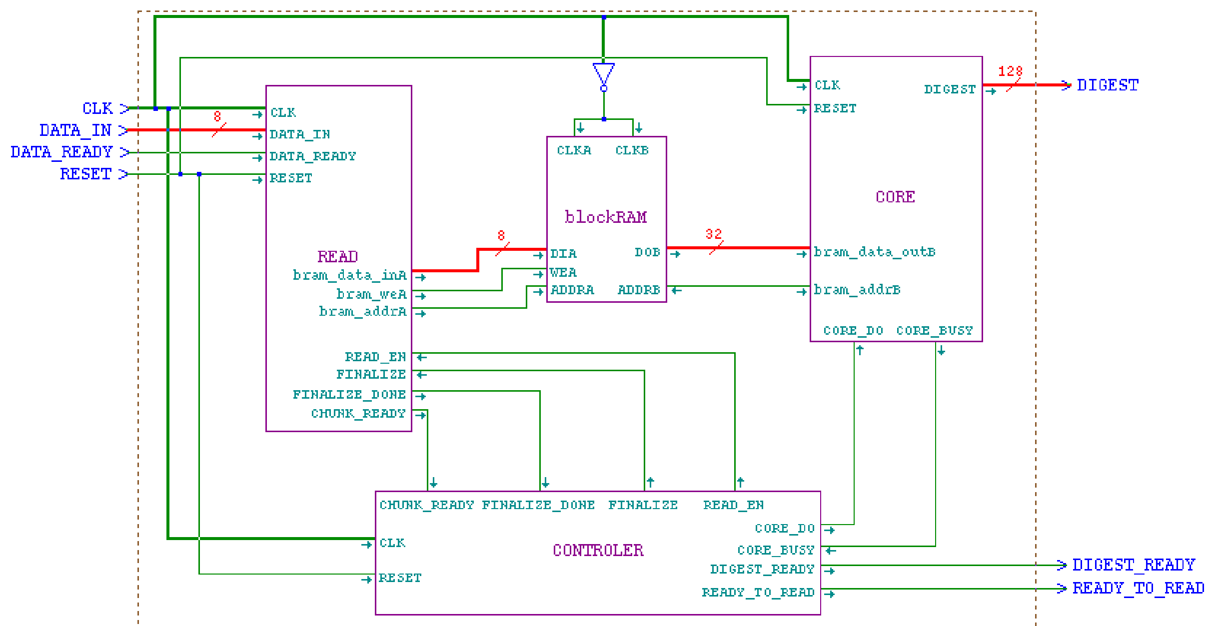
Další proces je již aktivován synchronně s hodinovým signálem. Při své aktivaci porovná poslední index dat ve vstupní frontě s jeho aktuální hodnotou. Při jejich nerovnosti a při schopnosti entity md5 číst data, přečte ze vstupní fronty další znak a provede jeho dešifrování v rámci přenosového protokolu. V případě, kdy minulý znak nebyl znakem ESC ($1B_{16}$), jej přivede na vstup entity md5.

Vzhledem k tomuto postupu a faktu, že ke čtecí indispozici dochází jednou za 64 přenesených znaků. Není využit signál ACK a je spoléháno jen na signál READY_TO_READ. Použití signálu ACK již zapříčinilo nefunkčnost dekódování vstupního protokolu z důvodu časové náročnosti porovnávání bitových vektorů. Nutno však podotknout, že tento signál svoji pozici v entitě neztratil, jelikož by byl důležitý při použití jiných přenosových protokolů, například Xmodem, který pracuje s velkými datovými bloky, kdy by k přenosu mohlo docházet 128 krát těsně za sebou. V takovém případě by se signál ACK využil a navíc by nebyl zdržením pro dekódování dalších dat.

Poslední proces obstarává přenos získaného otisku. Aktivuje se při náběžné hraně hodinového signálu a svoji činnost začne provádět od chvíle, kdy entita md5 nastaví signál DIGEST_READY na logickou 1. Od této chvíle začne postupně přenášet získaný otisk buď po 8 bitech, nebo pomocí ASCII znaků hexadecimálních čísel.

4.6 Implementace entity md5

Vlastní entita MD5 se skládá ze 3 jednotek a připojené paměti blockRAM. Jmenovitě jednotka READ, starající se o příjem dat a jejich uložení v odpovídajícím tvaru, jádro algoritmu, jednotka CORE, provádějící jednotlivé kroky iteračního výpočtu a CONTROLER, neboli řadič synchronizující celou činnost entity. Jednotlivé části jsou zapsány behaviorálním popisem procesu. Obrázek 4.6 demonstruje strukturu entity MD5.



Obrázek 4.6: Struktura entity MD5

4.6.1 Jednotka CONTROLER

Činnost této části tvoří 9 stavový, ve zjednodušené verzi 6 stavový konečný automat. Ten je implementován pomocí 2 procesů, kdy 1. z nich zpracuje signály od dalších 2 komponent a dle nich přepne automat do odpovídajícího stavu. Druhý proces je aktivován změnou stavu a nastaví požadované signály pro další hodinový cyklus. Automat pracuje s následujícími stavy:

- init: Tento stav je nastavován při resetu a v okamžiku odeznění resetu je nahrazen stavem *reading*.
- reading: V tomto stavu je prováděno přijímání vstupních dat. Žádná další činnost není prováděna. V případě příchodu signálu CHUNK_READY se automat přepne do stavu *reading_comp*. Dále se automat přepne do stavu *padding* v případě aktivace signálu EOF, či *padding_comp,* když tento přijde zároveň se signálem CHUNK_READY.
- reading_comp: V tomto stavu pokračuje čtení vstupních dat, je však signálem CORE_DO aktivováno jádro, kterému je předán již hotový blok dat. V případě, kdy je načten další blok dat a doposud není zpracován předchozí, automat se přepne do stavu *computing1*. Z tohoto stavu se automat může též přepnout do stavu *padding_comp* v případě aktivace signálu EOF, či *padding*

při současné deaktivaci signálu CORE_BUSY. V případě pouhé deaktivace signálu CORE_BUSY se automat vrátí do stavu *reading.*

- computing1: V tomto stavu probíhá dokončení zpracování jednoho 512 bitového bloku dat. Během této doby běží pouze jednotka CORE. Ve chvíli kdy se signál CORE_BUSY vrátí do logické 0, je automat přepnut zpět do stavu reading_comp.
- computing2: Je obdobou computing1, ale po dokončení práce jádra se automat, podle stavu signálu FINALIZE_DONE přepne buď do stavu padding_comp, nebo computing_last.
- padding: V tomto stavu je prováděno zarovnání posledního datového shluku. Je nastaven signál FINALIZE na hodnotu logické 1, čímž je aktivována jednotka PADDING. Všechny další jednotky jsou neaktivní. Po příchodu signálu CHUNK_READY označující další hotový datový shluk je automat přepnut do stavu padding_comp v případě, kdy signál FINALIZE_DONE zůstává v logické 0, nebo computing_last v případě, že zarovnání je dokončeno (signál FINALIZE_DONE je v logické 1).
- padding_comp: V tomto stavu jsou signály CORE_DO a FINALIZE v logické 1, čímž jsou aktivovány příslušné jednotky. Při aktivaci signálu CHUNK_READY, je automat přepnut do stavu computing2. V případě, kdy však zároveň signál CORE_BUSY přejde do logické 0 signalizujíc konec činnosti jádra, automat zůstane v aktuálním stavu, popřípadě přejde do stavu computing_last při současné aktivaci signálu FINALIZE_DONE. Při samotném přechodu signálu CORE_BUSY do logické 0, je automat přepnut zpět do stavu padding.
- computing_last: Značí zpracovávání posledního bloku dat. Je aktivován pouze signál CORE_DO a po dokončení práce jádra se automat přepne do stavu write.
- writing: Tento stav značí ukončení práce celé entity je nastaven signál DIGEST_READY informující o platnosti výsledného otisku. V tomto stavu automat zůstává až do příchodu dalšího signálu RESET.

V této jednotce se nejvýrazněji projevuje rozdíl mezi zjednodušenou a plnohodnotnou verzí zjednodušená verze neobsahuje signály *reading_comp, padding_comp, ani computing_last*, jeho funkci zastává stav computing2. Tím se také velmi snižuje množství podmínek pro přechod mezi stavy a logika automatu je o poznání menší.

4.6.2 Jednotka READ

Tato jednotka je tvořena jedním procesem zajišťujícím dvě činnosti. První z nich je pouhé čtení a ukládání do paměti blockRAM, přičemž je počítána délka dat. Tato činnost je aktivována s náběžnou hranou hodinového signálu a signálem READ_EN nastaveným na hodnotu logické 1. V tomto stavu proces zkontroluje signál DATA_READY a v případě, že se i ten nachází v logické 1, je 8 bitový blok dat na vstupním portu uložen na aktuální adresu paměti blockRAM a ta je zvýšena o 1. Současně je též nastaven signál ACK na hodnotu logické 1. V případě načtení 512 bitů je nastaven signál CHUNK_READY.

Druhou činností je zarovnání posledního datového shluku a připojení bitové délky. Tato část opět postupně ukládá 8 bitové hodnoty do paměti blockRAM, žádná data však nečte. Místo toho jako prvních 8 bitů uloží hodnotu 10000000_2, čímž připojí koncovou 1 za načtená data, a dále přejde do stavu secondphase, pokud v 512 bitovém bloku je ještě alespoň 64 bitů volných na délku zprávy, nebo do stavu firstphase, kdy se již délka zprávy do bloku nevejde. Tato činnost je řízena jednoduchým 3 stavovým automatem, jehož stavy jsou následující:

- firstphase Tento stav znamená dokončení zpracovávaného shluku dat, přičemž se za data a přidanou logickou 1 již nevejde 64 bitů délky zprávy. Shluk bude tedy dokončen logickými 0, nastaví se signál CHUNK_READY na hodnotu logické 1 a poté se přejde do stavu seconphase.
- seconphase Tento stav znamená dokončení aktuálního shluku, které bude znamenat konec činnosti této jednotky. Tato fáze je tvořena multiplexorem řízeným aktuální pozicí, kterým je na vstup paměti blockRAM přiváděna buď odpovídající část délky zprávy, nebo vektor osmi 0. Při úspěšném dosažení konce 512 bitů se opět nastaví signál CHUNK_READY na hodnotu logické 1 a se automat přepne do stavu finalizedone.
- finalizedone V tomto stavu se nastaví signál FINALIZE_DONE na hodnotu logické 1 a automat v něm přetrvává až do příchodu dalšího RESETu.

Část procesu obstarávající čtení příchozích dat je opakovaně spouštěna do chvíle, kdy řadiči nepřijde signál o konci vstupních dat. V případě, že je volána ve chvíli, kdy nejsou připraveny data (a signál DATA_READY je nastaven správně na hodnotu logické 0) neprovede se žádná činnost a tedy nedojde žádné nekonzistenci načtených dat. Stejná situace nastane při nadbytečném spouštění sekce pro zarovnání i v době, kdy je její činnost hotova (FINALIZE_DONE se nachází v logické 1).

V případě, že je proces aktivován a z jakéhokoli důvodu nedojde k načtení dat a to i v případě, kdy není sekce pro čtení volána, je automaticky nastaven signál ACK na hodnotu logické 0. Signál

ACK se tedy nachází v logické 1 vždy právě jeden takt po úspěšném načtení 8 bitového bloku z vnějšího zdroje a lze jej například využít k posouvání indexu vstupních dat.

Aby nedocházelo ke konfliktu zápisu do paměti a jejím čtením jádrem se po vytvoření lichého 512 bitového bloku dat nevrací ukazatel na začátek paměti, ale pokračuje ve svém zvyšování. Až po dokončení sudého bloku dat je ukazatel vynulován. Toto je jediný rozdíl mezi plnohodnotnou a zjednodušenou verzí, kdy u zjednodušené dochází k nulování po každém vytvořeném bloku. Toto není samozřejmě podmínkou, ale logika rozlišující sudý a lichý průběh zde postrádá smysl.

Data do paměti blockRAM jsou ukládána jako číslo ve formátu little endian. Díky tomuto odpadá nutnost, ať už předem, či po opětovném načtení, převádět data do tohoto formátu.

4.6.3 Jednotka CORE

Tato jednotka je tvořena opět jedním procesem, mohla by být však rozdělena do 2 procesů, kdy by druhý z nich implementoval konečný automat, který je nyní součástí procesu prvního. Neprobíhá-li výpočet, jednotka se aktivuje při náběžné hraně hodinového signálu a signálu CORE_DO ve stavu logické 1. Probíhá-li zpracování předchozího 512 bitového bloku dat. Je signál CORE_DO ignorován a s náběžnou hranou hodinového signálu je vždy provedena jedna iterace výpočtu. Jak již bylo řečeno, celý proces je ovládán konečným automatem majícím 3 stavy:

- loading V tomto stavu proces čeká na signál CORE_DO, při jeho aktivaci nastaví indexové proměnné a inicializuje 4 32 bitové proměnné a, b, c, d na obsah jednotlivých části aktuálního kontextu. Dále na adresový port paměti blockRAM přivede adresu prvního 32 bloku, nastaví signál CORE_BUSY informující o zatíženosti jádra na hodnotu logické 1 a přepne automat do stavu *computing*.
- computing V tomto stavu probíhá samotný výpočet. Je přečten 32 bitový blok z paměti blockRAM, vypočtena logická funkce, výsledky sečteny, upraveny indexové proměnné a nastavení další adresy pro příští čtení z paměti blockRAM. Jednotlivé úkony jsou uspořádány, aby je bylo možno provést v co nejkratší době, přičemž byla co nejdříve známa adresa do paměti. V tomto stavu proces zůstává po dobu 64 taktů odpovídajících 64 iteracím hašovacího algoritmu MD5. Po uplynutí 64 taktů se automat přepne do stavu *writing*.
- writing Tento stav označuje konec zpracování jednoho 512 bitového bloku dat. Hodnoty 4 32 bitových proměnných představujících výsledek této činnosti jsou přičteny ke kontextu uloženému v registru HASH a signál CORE_BUSY je nastaven na hodnotu logické 0 a automat je přepnut do stavu *loading*.

Hodnota 128 bitového vektoru HASH je vždy převedena do formátu little endian a přivedena na výstupní port DIGEST.

I v této jednotce platí, že jediný rozdíl mezi plnohodnotnou a zjednodušenou verzí tkví v rozlišení sudého a lichého běhu, kdy v plnohodnotné verzi je při sudém běhu k adrese do paměti přičteno 16, což je velikost jednoho 512 bitového bloku.

4.6.4 BlockRAM

Poslední částí entity MD5 je paměť blockRAM, ke které je přístup realizován samostatnou komponentou. Její použití bylo vynuceno z důvodu příliš velké režie na uchovávání 512 bitového bloku v registru. Bohužel jejím použitím se výrazně prodloužila kritická cesta a snížila se tak maximální frekvence odhadovaná syntetizátorem na asi 23MHz. Použití paměti ale velmi snížilo režii nutnou pro práci s načtenými daty.

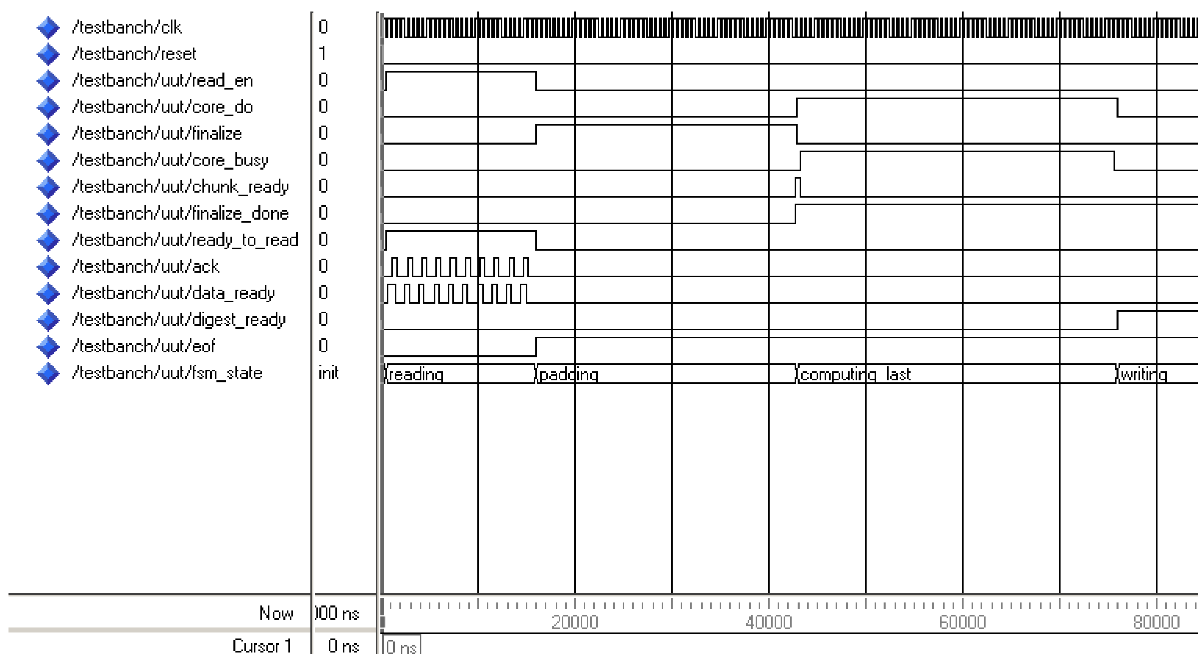
V návrhu byla použita dvouportová paměť RAMB*16_S9_S36*, ve skutečnosti je však využíváno pouze 8 bitů jednoho portu a 32 bitů portu druhého, paměť tedy není optimálně využita, ale její kapacita je výrazně vyšší, než je třeba. Dále jsou využity datové signály (signály DOA a DIB), signály pro zápis adres a u portu A signál pro aktivaci signálu(WEA). Signály ENA a ENB jsou připojeny na logickou 1 a ostatní signály jsou buď přivedeny na logické 0, nebo v případě výstupních signálů nejsou připojeny vůbec. Oba porty jsou připojeny na stejný zdroj hodinového signálu, který je přiveden přes invertor. Toto je opět z důvodu potřeby dostatečného předstihu připojení dat na signály před jejich přijetím. Vzhledem k tomuto faktu je jednotka CORE optimalizována, aby zápis adresy byl prováděn před polovinou jedné iterace a byl tím pokud možno využit celý takt hodinového signálu. Zápis přijatých dat do paměti je velmi jednoduchý a zarovnávání dat, kdy je též do této paměti zapisováno je psáno s ohledem na co nejrychlejší průběh, bohužel však právě zde je pravděpodobně nejvýraznější zdržení.

První port je připojen na jednotku READ. Jím jsou tedy do paměti nahrávána příchozí data pomocí 8 bitového signálu. Tímto portem probíhá pouze zápis do paměti. Druhý port je připojen na jednotku CORE. Ta na druhou stranu data pouze čte a to po 32 bitových blocích.

4.6.5 Časování entity MD5

Časování celé entity je rozděleno na vzestupné a sestupné hrany hodinového signálu. Jednotka CONTROLER je aktivována při sestupné hraně hodinového signálu, zatímco jednotky READ a CORE jsou aktivovány při vzestupné hraně hodinového signálu. Je tedy zajištěn až půlperiodový předstih signálu před jeho čtením. Poslední část celé entity, paměť BlockRAM je také aktivována při sestupné hraně hodinového signálu. Jelikož tato paměť se nachází mezi jednotkami READ a CORE, je tímto zaručen opět dostatečný předstih nastavení signálů před jejich čtením

bez jakéhokoli omezení a dalších prodlev. Při návrhu ostatních komponent je však na tuto skutečnost dobré brát zřetel pro využití celé periody hodinového signálu. Práce s pamětí je tedy prováděna co možná nejdříve při každé aktivaci procesu, aby zbytek logických operací mohl být prováděn i při nízké periodě hodinového signálu, kdy již paměť zpracovává zadaná data. Obrázek 4.7 znázorňuje průběh signálů při přenosu 10 bytů rychlostí 1MiB/s s frekvencí 2MHz.



Obrázek 4.7: Ukázka časování entity MD5

4.7 Konečné nastavení a vlastnosti

Vytvořený design je nastaven pro přenos dat rychlostí 57600 b/s s lichou paritou, jedním stop bitem a šířkou přenášených dat 8b. Nejvyšší rychlost není využita z důvodu kompatibility s přiloženou aplikací. Přesto je však plně funkční při všech rychlostech 38400 b/s až 921600 b/s, která je nejvyšší dosažitelná použitým přenosovým rozhraním. Jedinou výjimku tvoří rychlost 115200 b/s, kdy z neobjasněných příčin vzniká chyba zřejmě při dekódovaní přenosového protokolu. Tuto chybu se mi nepodařilo odstranit, přesto však se týká pouze této přenosové rychlosti.

4.8 Propustnost entity MD5

Teoreticky zpracování jednoho 512 bitového bloku dat trvá 66 taktů a jeho načtení 64 taktů. Tyto činnosti ale probíhají paralelně, tedy při větších objemech dat se náročnost výpočtu teoreticky blíží hodnotě 66 taktů na 64B dat plus koncové zarovnání. Výsledná maximální propustnost se tedy

při velkých objemech dat blíží $f \cdot \frac{64}{66}$ kde f je taktovací frekvence entity. Zjednodušená verze je pomalejší a její propustnost by se měla při velkých objemech dat blížit hodnotě $f \cdot \frac{64}{64+66}$. Hodinový signál SMCLK je nastaven na hodnotu 7.3728MHz. Teoreticky by průchodnost měla být asi téměř 7,1MiB/s u rychlejší a 3,6MiB/s u pomalejší verze. Při nízkých objemech dat je nutno brát v úvahu koncové zarovnání a u paralelní verze fakt, že při čtení prvních 512 bitů dat není prováděn výpočet a jejich zpracování tedy trvá 130 taktů místo 66.

Prakticky jsem propustnost otestoval generováním vstupních dat přímo v designu. Na vstup entity md5 byly přivedeny znaky 41_{16} a při každém příchodu signálu ACK byl zvednut čítač o 1. Testy na menších objemech dat jsem ověřil funkčnost a následně jsem nastavil horní hranici čítače na hodnotu 262144000. Těchto 250MiB dat zpracovávala plnohodnotná verze entity asi 36s a zjednodušená asi 71s. Tento výsledek odpovídá 7MiB respektive 3,5MiB a potvrzuje předpoklad získaný výpočtem.

5 Demonstrace funkce a výsledků

V této kapitole uvedu konečné možnosti, jak s výsledným designem komunikovat a získat požadované výsledky. Zároveň zde uvedu mnou testované vstupy společně se získaným otiskem. V případě, kdy daný vstupní řetězec testuje konkrétní vlastnost nebo stav, tuto skutečnost uvedu.

5.1 Přiložená ukázková aplikace

Pro ukázku komunikace a činnosti je vytvořena jednoduchá aplikace posílající soubor přes sériový port a dekódující příchozí otisk. Komunikace probíhá rychlostí 57600 bitů/s s lichou paritou, jedním stop bitem a datovou šířkou 8 bitů. Při startu aplikace je možné si pomocí menu *Port* vybrat sériové rozhraní COM1 - COM9, skrze které bude probíhat přenos. Dále v menu *Soubor* je možné otevřít požadovaný soubor. Po otevření souboru se zahájí přenos a po jeho dokončení je do okna aplikace vypsán otisk ve formě hexadecimálních číslic. Jednotlivé otisky jsou vypisovány pod sebe. Při špatné konfiguraci aplikace, kdy se přenos nezdaří se po chvíli objeví upozornění *Timeout* a je možné s aplikací dále pracovat. Celou aplikaci je možné ukončit zavřením jejího okna, či výběrem položky *konec* v nabídce *Soubor.* Tento program pracuje s binární podobou výstupního otisku a neumí jej správně zobrazit pokud je použit textový výstup.

5.2 Použití programu Hyperterminal

Ke komunikaci je možné též použít terminálovou aplikaci pro komunikaci a přenos souborů přes sériové rozhraní, například program Hyperterminal. Tento program obsahuje funkci *Přenos textového souboru*, jíž lze v tomto případě s úspěchem využít. Tato funkce však přenese zadaný soubor bez jakýchkoli dalších informací. Z tohoto důvodu je třeba vstupní soubor upravit, aby odpovídal přenášenému protokolu, viz kapitola 4.5. Je tedy nutné za konec souboru vložit znak 04_{16} a tím označit konec přenášených dat. Dále v případě, že soubor obsahuje znaky 04_{16} nebo $1B_{16}$, je nutné vytvořit escape sekvence, tedy před tyto znaky umístit znak $1B_{16}$, v opačném případě nebude soubor zpracován korektně, bude předčasně ukončen, nebo nebudou zpracovány znaky $1B_{16}$. V případě výskytu znaku $1B_{16}$ na samém konci souboru, tedy před přidaným znakem 04_{16}, nebude rozpoznán konec vstupních dat.

Při použití tohoto nebo podobného terminálu je též žádoucí nastavit textový výstup v designu odkomentováním a zakomentováním příslušných řádků, jež jsou v kódu vyznačeny.

5.3 Otisky získané z vybraných vstupů

Pro ověření správné funkce designu jsem použil různá testovací data, která prověřují různé stavy, do kterých se funkce může dostat. Výsledný otisk jsem ověřoval podle referenční implementace uvedené v RFC 1321 [1], programem mddriver. všechny získané otisky se shodovaly s výstupy referenční implementace.

Vstupní data	Cíl testování	Získaný otisk
Prázdný řetězec	Žádná příchozí data	D41D8CD98F00B204E9800998ECF8427E
Znak „A“	Běžný krátký vstup	7FC56270E7A70FA81A5935B72EACBE29
„ABCDEFGHIJKLMNOPQR STUVWXYZ“	Běžný krátký vstup	437BBA8E0BF58337674F4539E75186AC
„ABCDEFGHIJKLMNOPQR STUVWXYZABCDEFGHIJ KLMNOPQRSTUVWXYZA BCDEFGH“	Funkce dvoufázového zarovnání	3BA2EC139C4D72C68E7F5D6F05462548
128 krát opakující se anglická abeceda (velká písmena)	Delší vstup	9493DD69B3E03BC4F1A64DEFC1221F70
Soubor mddriver (Linux ELF32 binární soubor)	Dlouhá binární data	58E773F9E877A294025C2B0ABE8FED1A

Tabulka 5.1: Testování funkčnosti na vzorcích dat

6 Možnosti dalšího rozšíření

Tato práce jistě není absolutní implementace daného problému. Spíše jde o funkční řešení, které může být mnoha způsoby rozšiřováno.

6.1.1 Převod zápisu do strukturální podoby a optimalizace

Jednou z nejvýraznějších slabin současného návrhu je jeho behaviorální podoba. Některé vlastnosti musely být oželeny z důvodu asi 3% přesahu kapacity čipu. Přepsáním jednotlivých částí do strukturální podoby by se zcela jistě ušetřilo mnoho logických bloků. Dále by se jistě i v tomto návrhu našla možnost jej nějakým způsobem optimalizovat. Moje snaha byla v tomto ohledu maximální, ovšem zcela jistě by se daly najít drobnosti, jenž by mohly pracovat lépe.

6.1.2 Jiná koncová aplikace

Přiložená aplikace sice hezky demonstruje činnost, ale funkčně je omezená a použitá komponenta je relativně zastaralá a nepodporuje novější verze prostředí. Toto činní aplikaci špatně rozšiřitelnou, navíc je kvůli ní omezena přenosová rychlost.

6.1.3 Jiné rozhraní

Velkým přínosem by byla implementace jiného přenosového rozhraní. Podle testů jsem zjistil, maximální propustnost entity MD5 na současných 7,3MHz asi 7MiB. Nejvyšší rychlost rozhraní RS232 je však pouze necelý 1Mib.

7 Závěr

Cílem této práce bylo vytvořit v jazyce VHDL kompaktní design realizující algoritmus MD5 v programovatelném hradlovém poli na výukové platformě FITkit. Tohoto cíle bylo dosaženo a byl vytvořen design komunikující po sériovém rozhraní RS232 za využití jednoduchého přenosového protokolu s počítačem a generující otisk daným hašovacím algoritmem.

Při vlastní realizaci jsem narazil na kapacitní možnosti použitého čipu FPGA, Z tohoto důvodu jsem původní návrh omezil a vytvořil zjednodušenou verzi, která pracuje spolehlivě a dostatečně rychle. V závěru samotné práce se mi ale podařilo původní návrh optimalizovat do takové míry, že již s jeho nasazením nebyl žádný problém, přesto však zabírá téměř celý čip a občas z nevyjasněných příčin syntetizátor vygeneroval příliš velký konfigurační soubor. Z tohoto důvodu jsem do této práce tedy zahrnul obě verze s důrazem na plnohodnotnou a dvakrát rychlejší verzi.

Po problémech s kapacitou čipu byl vytvořen plně funkční design, který se může stát součástí rozsáhlejšího projektu, stejně jako může sloužit pro demonstraci funkce algoritmu MD5 a jeho implementaci v jazyku VHDL pro programovatelné hradlové pole FPGA.

Literatura

[1] *Request for comments 1321* [online]. [cit. 14.5.2008] URL: <http://www.ietf.org/rfc/rfc1321.txt?number=1321>

[2] *Platforma FITkit* [online].Fučík, Oto. [cit. 14.5.2008] URL: <http://merlin.fit.vutbr.cz/FITkit/>

[3] Wilson, P. R. *Design Recipes for FPGAs.* Elsevier 2007. ISBN 978-0-7506-6845-3

[4] *Wikipedia* [online]. [cit. 14.5.2008] URL : <http://www.wikipedia.org>

[5] *Xilinx Spartan-3 FPGA family: Complete data Sheet*. Xilinx®. [cit. 14.5.2008] URL: <http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf>

Seznam příloh

Příloha 1. CD se zdrojovými texty a přiloženou aplikací.