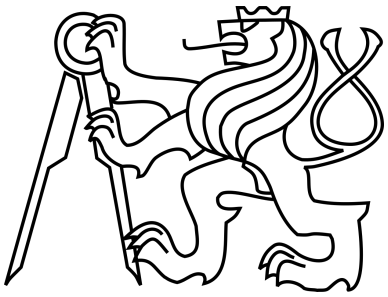
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra mikroelektroniky



Diplomová práce

Návrh paralelního soft-procesoru

Adam Patera

Vedoucí práce: Ing. Stanislav Vítek, Ph.D.

Studijní program: Komunikace, multimédia a elektronika

Obor: Elektronika

9. ledna 2018

Poděkování

Tímto bych rád poděkoval Ing. Stanislavovi Vítkovi, Ph.D. za odborné vedení mé diplomové práce.

Čestné prohlášení

Prohlašuji, že jsem zadanou diplomovou práci vypracoval sám s přispěním vedoucího práce a používal jsem pouze literaturu v práci uvedenou. Dále prohlašuji, že nemám námitek proti půjčování nebo zveřejňování mé diplomové práce nebo její části se souhlasem katedry.

V Praze dne 9. 1. 2018 .....

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Patera** Jméno: **Adam** Osobní číslo: **393140**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Komunikace, multimédia a elektronika**

Studijní obor: **Elektronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh paralelního soft-procesoru

Název diplomové práce anglicky:

Design of Parallel Soft-processor

Pokyny pro vypracování:

1) Prostudujte metody paralelizace procesorových jednotek.
2) Navrhněte a implementuje na FPGA paralelní počítač se 4 procesorovými jednotkami. Navrhněte vhodný model sdílení zdrojů.
3) Pro složitější matematické operace implementujte matematický koprocesor.
4) Implementujte ASM překladač. Na vhodných úlohách demonstrujte možnosti architektury.
5) Diskutujte efektivitu návrhu a možnosti rozšiřitelnosti. Porovnejte s jinými architekturami.

Seznam doporučené literatury:

[1] Patterson D. A., Hennessy J. L. Computer Organization & Design, 5th Edition. Elsevier, 2014. ISBN: 978-0-12-407726-3
[2] Gu Ch. Building Embedded Systems: Programmable Hardware. Apress, 2016. ISBN 978-1484219188

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Stanislav Vítek, Ph.D., 13137

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2017** Termín odevzdání diplomové práce: **09.01.2018**

Platnost zadání diplomové práce: **10.09.2018**

Ing. Stanislav Vítek, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání	Podpis studenta

Abstract

The primary goal of this diploma thesis was to design a multi-processor computer with dedicated mathematical coprocessor and implement it using Field Programmable Gate Array, or FPGA. In addition, a custom assembler for the computer has been developed along with some other software utilities. Proposed computer thus incorporates four independent execution units and universal CORDIC processor. Final chapters of the work are then focused on implementing VGA and PS/2 controller.

Keywords: Soft-microprocessor, CORDIC, UMA

Abstrakt

Diplomová práce se zabývá návrhem syntetizovatelného víceprocesorového systému včetně dedikovaného matematického koprocesoru na architektuře FPGA. Součástí práce je i implementovaný překladač jazyka symbolických adres *assembler*. Počítač tak obsahuje čtyři nezávislé výkonné jednotky a univerzální CORDIC procesor. V závěru práce je věnována rovněž pozornost návrhu integrovaného ovladače VGA videa a PS/2 rozhraní.

Klíčová slova: Soft-mikroprocesor, CORDIC, UMA

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Předmluva

Předmětem práce byl návrh víceprocesorového systému s integrovaným matematickým koprocesorem spolu se základními softwarovými nástroji a to výhradně překladače jazyka symbolických adress (*assembler*). Cílový chip je Spartan-6 LX16 a k návrhu byla užita vývojová deska Nexys 3^{TM}. Práce je přímou realizací možného rozšíření diskutovaného v původní bakalářské práci[12]. Od samého začátku byl návrh cílen především na jednoduchost implementace a to jak z důvodu časové náročnosti projektu tak i velikosti plochy cílového chipu. Zároveň byl však dodržen požadavek na praktickou využitelnost celého výpočetního zařízení. Soft-procesor je stále schopen operovat na poměrně vysokém kmitočtu 100 MHz, počítat složité matematické funkce a komunikovat s okolním prostředím prostřednictvím integrovaných periférií.

1.2 Poznámky k notaci a členění práce

Text je intuitivně členěn do kapitol a oddílů a to pokud možno v chronologickém pořadí. Součástí této kapitoly je stručný rozbor výpočetní paralelizace a formátu reálných čísel nutného k návrhu matematického koprocesoru. Nalezneme tu rovněž rychlý souhrn původního návrhu[12], ze kterého tato práce vychází. Ve druhé kapitole již nalezneme detaily kompletního návrhu víceprocesorového počítače. Třetí kapitola se pak zabývá návrhem matematického koprocesoru a rovněž zde můžeme najít vyčerpávající přehled a definice všech dostupných instrukcí koprocesoru. Předmětem čtvrté kapitoly jsou integrované periférie, tzn. ovladač videa a klávesnice.

V páté kapitole je popsán imlementovaný překladač jazyka symbolických adres spolu se softwarovým emulátorem celého počítače. V šesté kapitole jsou uvedeny výsledky nejdůležitějších automatizovaných testů koprocesoru

spolu s ukázkovým programem počítače. V závěrečné kapitole pak nalezneme krátké zamyšlení nad možnými úpravami či vylepšeními architektury.

- Všechna bitová pole instrukcí, registrů, paměťových slov apod. jsou číslována od LSB (*Least Significant Bit*) do MSB (*Most Significant Bit*), kde LSB se vyskytuje vždy na pravé straně bitové posloupnosti a MSB na levé. Počítač obecně implementuje *Little Endian* stanard.

- V závislosti na kontextu někdy označujeme slovem *počítač* celý návrh, jindy pouze samostatný systém jednotek MAIN, AUX1, AUX2 a AUX3 – a to v případě, že je chceme pro názornost explicitně odlišit od koprocesoru FXP.

- Vzhledem k nepřeložitelnosti některých termínů a mnemonik bude obsah tabulek ponechán v anglickém jazyce. Jedná se například o názvy bitových polí v registrech či instrukcí, kde daný symbol přímo odpovídá plnému názvu. Dále v textu, pokud to bude možné, budou zachovány české termíny následované původním anglickým významem v závorce.

1.3 Metody výpočetní paralelizace

V tomto oddíle budou stručně rozebrány základní principy výpočetní paralelizace. Vzhledem k značné rozsáhlosti dané problematiky se zde omezíme jenom na popis základních mechanismů, které jsou přímo relavantní k diskutovanému návrhu počítače.

Zcela obecně můžeme rozlišit několik specifických technik výpočetní paralelizace:

- Paralelismus na úrovni instrukčního slova (*Subword Parallelism*)

- Pipelining

- Víceprocesorové systémy

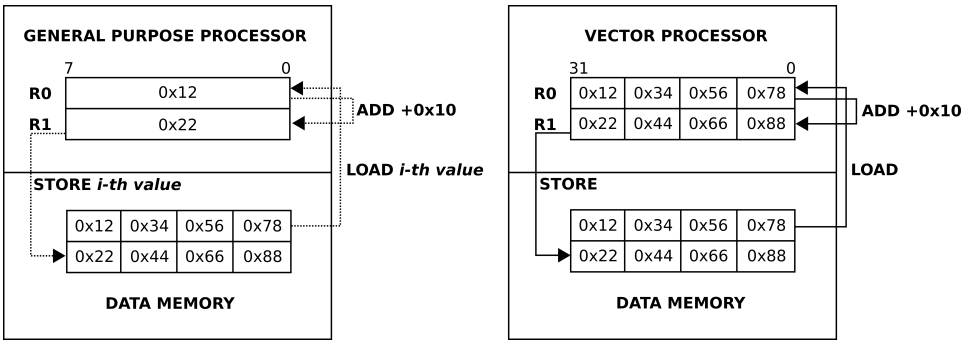
Poznámka. Všechny výše uvedené koncepty jsou často implementovány současně a to v závislosti na účelu a nárocích návrhu.

1.3.1 Subword Parallelism

Software specificky navrhnutý pro práci s velkým množstvím dat bude obecně velmi náročný na výpočetní prostředky. Takovýto program bude často provádět identické a repetetivní výpočty na velké množině vstupních dat. Namísto sekvence stejných operací by tedy bylo velmi výhodné navrhnout systém tak, aby mohly být provedeny současně ve stejném instrukčním cyklu. Klíčovým

konceptem je zde implementace *registrového souboru* (register file) jehož individuální registry mají bitovou šířku rovnou násobku nativního datového slova procesoru spolu s replikací funkčních jednotek počítače, jako je například ALU. Jednotlivé registry souboru se navenek chovají jako *regulerní* registry s tím rozdílem, že architektura počítače si je *vědoma* jejich speciálního významu pro některé z dedikovaných instrukcí instruční sady. Ve skutečnosti jsou totiž registry rozděleny do tzv. *půlslov* (half-word). Toto přímo odpovídá tomu, jak je jediný dlouhý bitový řetězec uložený v registru interpretován.

Předpokládejme nyní fiktivní počítačovou architekturu s 16-bitovou nativní délkou datového slova. Souborový registr by mohl obsahovat registry čtyřnásobné délky, tedy 64 bitů. Procesor by pak typicky vyčetl jedinou instrukci z programové paměti, která by umožnila zkopírovat celých 64 bitů do takového registru najednou. Namísto čtyř *load* instrukcí tak byla provedena pouze jedna jediná. Na rozdíl od *regulerního* registru zde každé 16-bitové slovo představuje zcela nezávislou hodnotu. Efektivně lze na takový registr nahlížet jako na vektor a obecně této technice říkáme *Single instruction, multiple data* (SIMD) *vektorový* počítač. Další z instrukcí pak může provést součet, umocnění či jinou aritmetickou operaci na jednotlivých komponentách vektoru s užitím replikovaných funkčních jednotek. V závislosti na architektuře mohou být operace prováděné v daném instrukčním cyklu na jednotlivých vektorových komponentách zcela rozdílné – například na nižších dvou komponentách vektoru může být proveden součet, zatímco vyšší dvě komponenty vektoru jsou umocněny.



Obrázek 1.1: Univerzální a vektorový procesor

- **Univerzální CPU.** Příklad hromadného součtu. Univerzální procesor vyčte jediné slovo z datové paměti, přičte k němu konstantu a výsledek uloží do cílového registru. Obsah tohoto registru je pak zapsán zpět do datové paměti. Tento proces je opakován celkem čtyřikrát ; jednou pro každé z čísel.

- **SIMD CPU.** Vektorový procesor vyčte všechna čtyři zdrojová slova z datové paměti najednou a uloží je do registrového souboru. Součet je proveden konkuretně na všech slovech registru a výsledek je najednou v jediném kroku zapsán zpět do datové paměti.

K technologii SIMD se však rovněž váže řada designových nástrah a problémů. Ta nejdůležitější, a rovněž nejvíce patrná, je skutečnost, že ne každý algoritmus lze snadno *vektorizovat*, neboli rozdělit na nezávislé a *konkurentní* datové manipulace. Ve skutečnosti jde o extrémně komplikovaný úkol a proto rovněž univerzální automatizované nástroje (kompiler programovacího jazyka vyšší úrovně) často nevygeneruje žádné SIMD instrukce a namísto toho zvolí přísně sekvenční přístup při manipulaci s daty. Toto má za důsledek degradaci výkonu a hlavně plýtvání plochou chipu, neboť dané replikované funkční jednotky a další hardware zůstává nevyužit. Software napsaný kompletně lidským operátorem tak může být velmi náročný na vývoj.

Plnění vektorových registrů větší šířky[1] daty je rovněž relativně komplikované a k optimální funkci často vyžaduje jisté permutace datových přesunů v systémové paměti a univerzálních registrech. SIMD kód rovněž nelze snadno přesunovat mezi cílovými platformami, neboť užití vektorových instrukcí je přísně vázáno na danou architekturu. Pokud není podporována vhodná instrukční sada, pak jedinou možností kompileru je vygenerovat tradiční sekvenci SISD (*Single instruction, single data*) instrukcí. SIMD hardware rovněž klade větší nároky na plochu chipu a celkovou spotřebu, a to z důvodu již výše uvdených velkých registrových souborů a replikovaných funkčních jednotek. Tato nevýhoda je ještě umocněna a zesílena pokud nelze vygenerovat příslušný SIMD program.

Poznámka. Vektorové procesory nacházejí nejširší uplatnění právě u vysokorychlostního zpracování videa, které vyžaduje mnoho identických a vzájemně nezávislých operací na velké množině dat (maticové operace apod.). Navíc je v softwaru grafických procesorů o poznání méně prostoru pro větvící instrukce, což dále zvýrazňuje sílu SIMD (případně MIMD[2]) techonologie v porovnání s univerzálními procesory.

1.3.2 Pipelining

Pipelining je mechanismus, který umožňuje *po částech konkurentní* vykonávání instrukcí a to s využitím jediné výkonné jednotky. Obecně pipelining zvyšuje *propustnost* instrukcí, zatímco doba výkonu dané instrukce zůstává nezměněna. Pokud bychom nahlíželi na posloupnost instrukčních cyklů

[1] Například 128–bitové MMX a SSE registry implementované x86 kompatibilní architekturou.

[2] *Multiple Instructions, Multiple Data.* Identický princip, nyní však instrukce operují na datech paralelně, nikoliv sekvenčně jako u SIMD.

jako na jakýsi pomyslný *informační* tok, pak pipelining jednoduše zvyšuje šířku pásma paralelním vykonáváním částí instrukčního cyklu jednotlivých instrukcí. Instrukční cyklus obecně sestává z následujících fází:

1. **IF.** Instruction fetch
2. **ID.** Instruction decode
3. **EX.** Instruction execution (preceeded by operand read)
4. **MA.** Memory access
5. **WB.** Writeback stage

Výčet je chronologický, tzn. instrukční cyklus začíná krokem 1 (fetch) a končím krokem 5 (writeback).

Poznámka. Pro názornost jsme uvedli přístup do paměti jako seperátní fázi instrukčního cyklu, neboť vzhledem k časové ose lze pak lépe pozorovat činnost a pořadí jednotlivých událostí v pipeline. Některé instrukce této specifické fáze využívají, jiné ne.

Na kontrolní jednotku procesoru lze nahlížet jako na konečný stavový automat, FSM[3], kde každý stav představuje příslušnou fázi instrukčního cyklu. Pro účely *pipelnování* je klíčové si povšimnout, že pro každý ze stavů FSM bude vždy vygenerován *veškerý* potřebný hardware nezávisle na *současném* stavu automatu. Dle teorie[3] lze každý stavový automat zkonstruovat vytvořením excitační tabulky, kde jsou popsány veškeré vstupy a výstupy spolu s příslušnými přechodovými funkcemi. Deterministický stavový automat pak může být v daném okamžiku právě v jednom jediném stavu. Toto tvrzení, ačkoliv správné z matematického hlediska, již však neplatí pro fyzickou, hardwarovou implementaci.

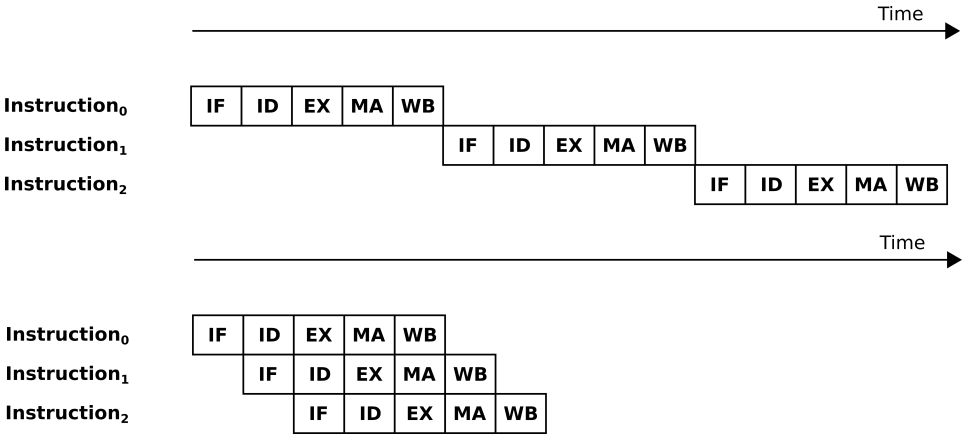
Jako hezký příklad nám může posloužit libovolný logický kombinační obvod. Je patrné, že obvod bude spojitě produkovat *okamžitý* výstup nezávisle na tom, zda jsou příslušné vstupy pokládány za *platné*. Toto je rovněž důvod existence logických hazardů v kombinační logice. Výstupy individuálních hradel se rovněž v průběhu operace obvodu mění dokud nedosáhnou *ustálené* hodnoty a proto je nutné logické funkce čistě kombinačních obvodů konstruovat tak, aby se navenek zamezilo metastabilitě výstupních signálů.

Většina logický obvodů je sekvečních a tudíž řízena hodinovým signálem. Pokud je součástí návrhu i některý z čistě kombinačních obvodů, jako například jednoduchá ALU, systém je synchronizován tak, aby v jednom hodinovém cyklu aplikoval *platné* signály na vstupní vodiče a platné výstupy

[3]FSM, Finite State Machine.

vyčetl v následujícím hodinovém cyklu. Přesný hodinový cyklus (tedy místo na časové ose) záleží na nejdelší kritické cestě pro daný hodinový kmitočet, nicméně z hlediska synchronizace alespoň jeden hodinový cyklus je z teorie nutno vyčkat, než může být výstup prohlášen za *platný*. Mezitím projde výstup obvodu mnoha stavy, ale v jednom přesně definovaném hodinovém cyklu bude prohlášen za platný a registrován. Přesně stejnou úvahu lze aplikovat na čistě sekvenční logiku.

V nepipelinovaném designu systém navštíví každý ze stavů automatu v instrukčním cyklu právě jednou. Po *fetch* stavu následuje přechod do *decode* stavu, nicméně hardware vygenerovaný pro *fetch* stav bude pokračovat ve vyčítání instrukčních slov z programové paměti nezávisle na současném stavu. Podobně jakmile je instrukce dekódována a přejde do *execute* stavu, dekódovací stav bude nadále pokračovat ve stejné operaci. Jediné co se v daném stavu automatu změnilo jsou vstupní signály, které jsou nyní považovány za *neplatné* a jsou kompletně ignorovány. Toto je zdrojem velké neefektivity, neboť příslušný hardware je stále fyzicky přítomen na chipu, ačkoliv v danou chvíli neslouží žádnému účelu a pouze zabírá cenné prostředky. Pokud bychom mohli zaručit, že každý ze stavů bude mít vždy platné vstupní signály (a tudíž platný výstup), systém by mohl provádět vícero instrukcí současně za jeden instrukční cyklus a problém by byl vyřešen.

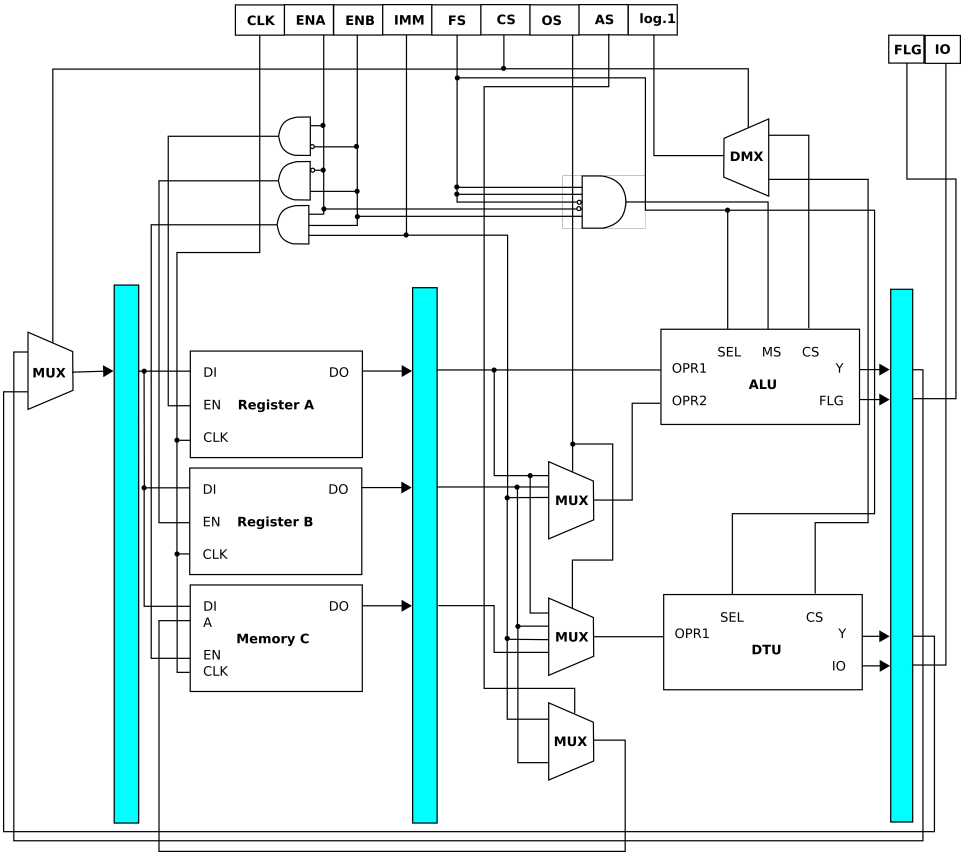


Obrázek 1.2: Časový průběh pipelinovaného a nepipelovaného instrukčního cyklu

Původní datová cesta výkonné jednotky počítače by musela být principu pipelinování patřičně přizpůsobena. Každou vyčtenou instrukci by bylo nutné *zvlášť* registrovat, stejně jako výstupy ALU a DTU funkčních jednotek a to před *writeback* fází daného instrukčního cyklu. Výstupy pipelinových registrů by pak musely být vedeny zpět jako vstupy do kontrolní jednotky, která by rovněž musela být novému principu přizpůsobena. Každý ze stavů

v (1.3.2) je pak nutné přepsat tak, aby byla v každém okamžiku zaručena a ověřena platnost vstupů z příslušně fáze pipelinového registru, což otevírá prostor pro celou řadu událostí, které je nutné řešit přidáním další komplexní logiky.

Jednotlivé fáze (umístění) pipeline registrů jsou podobně jako v [1] zvýrazněny pomocí modrých obdélníků níže.



Obrázek 1.3: Pipelinovaná datová cesta výkonné jednotky

Ve skutečnosti by implementovaná instrukční sada mohla být pipelinována relativně snadno, neboť byla navrhnuta s cílem co nejvyšší jednoduchosti a srozumitelnosti. Klíčovou roli zde hrají následující faktory:

1. Instrukční slova jsou všechna stejné délky a *okamžitá* (immediate) data jsou jeho integrální součástí.

2. Kódovací schéma instrukcí je symetrické a tudíž snadno dekódovatelné.

3. Počítač implementuje *Load & Store* architekturu.

Díky prvnímu bodu je velmi jednoduché vyčítat instrukce z programové paměti do první pipeline fáze, protože počet hodinových cyklů nutných k vyčtení instrukce je pokaždé konstatní, narozdíl od architektur s proměnným počtem slov. Druhý bod naznačuje, že zdrojové registry mohou být přečteny v průběhu dekódovacího procesu instrukce. Formát instrukcí je společný všem existujícím instrukcím v sadě a operandy jsou rovněž často implicitní. Kontrolní a pipeline logika tudíž nemusí pro určení operandů přímo dekódovat danou instrukci. Konečně třetí bod implikuje, že v systému existují pouze dva typy instrukcí (nepočítaje XTR instrukce), které mohou přistupovat do paměti. Díky této restrikci je možné využít *execute* fázi pro výpočet příslušných paměťových adres. Tento fakt dále zjednodušuje činnost pipeline, neboť je zaručeno, že žádna další instrukce nebude operovat na paměti. Toto je veliká výhoda vzhledem k tomu, že současný návrh navíc počítá s víceprocesorovým řešením což z důvodu nízkého počtu pracovních registrů dále navyšuje utilizaci datové paměti.

Dle výše uvedených poznatků by se mohlo zdát, že všechny instrukce lze jednoduše a poholdně pipelinovat. Ve skutečnosti však existuje řada okolností, které mohou činnost pipeline zcela zastavit (*stall*), neboli zamezit dokončení plného instrukčního cyklu instrukce. Tyto události nazýváme hazardy a rozlišujeme tři rozdílné typy.

- **Structural Hazard.** Toto jednoduše znamená, že některá z kombinací instrukcí je vzájemně nekompatibilní a nevhodná k pipelinování. Daná kombinace instrukcí není návrhem podporována.
- **Data Hazard.** Datový hazard nastává, když činnost pipeline musí být pozastavena (*stalled*), protože jedna fáze pipeline musí vyčkat na dokončení jiné. Toto je způsobeno vzájemnou závislostí instrukcí v pipeline – konkrétně závislost i-té instrukce na výsledku $i-1$-té instrukce. Uveďme jednoduchý příklad:

  ```
  add #3
  add b
  ```

  Je patrné, že výsledek druhé instrukce je závislý na první, neboť cílový operand první instrukce (implicitně registr A, akumulátor) je jedním ze zdrojových (a rovněž cílových) operandů druhé instrukce. Výše uvedený příklad datového hazardu by okamžitě pozastavil činnost pipeline a způsobil velikou degradaci výkonu. Instrukce ADD nezapíše svůj výsledek až do páté fáze (*write-back*) instrukčního cyklu, což znamená, že bychom museli pozastavit (*stall*) pipeline po dobu právě tří hodinových cyklů. Vzájemná závislost je velice častá a ve většině případů neřešitelná pomocí prostých permutací závislých instrukcí. Ze softwarového hlediska je tudíž tento problém neřešitelný. Namísto toho si lze uvědomit, že nemusíme nutně vyčkávat až do zpětného zápisu výsledku do

registrů, nýbrž můžeme odebrat výsledek součtu z výstupu ALU a ten připojit rovnou na vstup druhé instrukce v pipeline. Rovněž z tohoto důvodu je výsledek ALU na schématu(1.3) registrován. Této technice se v angl. říká **forwarding** nebo také **bypassing**.

- **Control Hazard.** Kontrolní, neboli také větvící (*branch hazard*) hazard, nastává ve chvíli, kdy je nutné učinit rozhodnutí na základě instrukce, která ještě nebyla dokončena. V případě, že v i-tém cyklu je zařazena do pipeline podmíněná větvící instrukce, existují nyní dvě instrukce, které mohou být pipelinovány v příštím $i+1$ cyklu. Nevyřešená podmínka větvení může být splněna – potom by měla být do pipelina zařazena instrukce na adrese cíle větvení, nebo může být nesplněna a v takovém případě musí být do pipeline zařazena hned instrukce následující[4]. Deterministickými metodami však nelze určit, zda podmínka větvení bude či nebude splněna a to je příčina *větvícího* hazardu.

Zatímco datový hazard lze vyřešit pomocí **forwarding** a **bypassing** technik, hazard větvení je mnohem náročnější na řešení. Přirozeně se nabízí nejjednodušší řešení, a to pozastavit činnost pipeline do doby, kdy je podmínka větvení známa. Takovýto postup je z praktického hlediska nepřijatelný, neboť by vedl k veliké degradaci výkonu, a to hlavně v *control-heavy* částech softwaru, jako jsou *switch* či vnořené *if-else* struktury. Jelikož neexistuje způsob, jak s pravděpodobností $p = 1$ určit zda dojde k větvení nebo ne, je nutné výsledek nějakým způsobem předpovědět. Správná predikce nezpůsobí pozastavení pipeline a umožní pokračovat ve zpracování instrukčního toku bez jakékoliv penalizace. Chybná predikce na druhou stranu vyústí v úplné vyprázdnění (*flush*) pipeline počínaje větvící instrukcí.

Predikátor větvení je komplikované zařízení, a proto zde pouze okrajově zmíníme základní principy predikce.

- **Veškerá větvení jsou považována za nevykonaná.** V případě správné predikce pokračuje pipeline plnou rychlostí bez jakkékoliv penalizace. Naopak v případě chybné predikce musí být celá pipeline vyprázdněna.

- **Pouze některá větvení jsou považována za nevykonaná.** Tento přístup předpovídá, že veškeré dopředné paměťové skoky nebudou provedeny a veškeré zpětně paměťové skoky budou provedeny. Tato heurestika vychází ze stereotypního schématu psaných programů, kdy zpětný skok na adresu typicky představuje smyčku, která bude s velikou pravděpodobnostní vždy vykonána.

[4]Případně obráceně, záleží, jestli daná větvící instrukce provádí podmíněný skok nebo vynechání instrukce (*skip*). Princip je však totožný.

- **Dynamická predikce.** Výše uvedené predikce jsou v čase neměnné a nijak nerespektují skutečný průběh programu. Dynamický prediktor je typicky konečný stavový automat, který průběžně analyzuje vykonávaný kód a přiřazuje pravděpodobnosti skoku jednotlivým větvícím instrukcím v závislosti na pozorované historii větvení.

Obrovský přínos pipelinování instrukcí je možnost vykonávat více instrukcí najednou v jednom hodinovém cyklu[5] a to bez jakékoliv změny v uživatelském softwaru. Mechanismus pipelinování je ze strany programátora zcela neviditelný.

1.3.3 Víceprocesorové systémy

Velmi obecně rozlišujeme tzv. těsně vázané (*tightly coupled*) a volně vázané (*loosely coupled*) systémy. Hlavním rozdílem mezi oběma systémy je paměťová organizace. Těsně vázané systémy sdílejí stejný paměťový prostor a často i přímo fyzickou paměť, zatímco volně vázané systémy dedikují privátní paměť každé procesorové jednotce. Data mezi procesory jsou pak přerozdělována pomocí některého ze známých mechanismů *zasílání zpráv* (message passing). Opět, vzhledem ke značné rozsáhlosti problému se dále v textu budeme zabývat pouze těsně vázanými procesorovými systémy.

- Adresový prostor je viditelný a vzájemně sdílen všemi procesory v systému.
- Dva a více procesorů všechny sdílejí stejnou sběrnici pro přístup do paměti. Pokud je sběrnice obsazena zatímco některý z procesorů vyžaduje přístup do paměti, pak je nucen vyčkat. Doba čekání je závislá na implementovaném mechanismu sdílení sběrnice.

Nejjednodušší víceprocesorové systémy sdílejí pouze jedinou sběrnici.

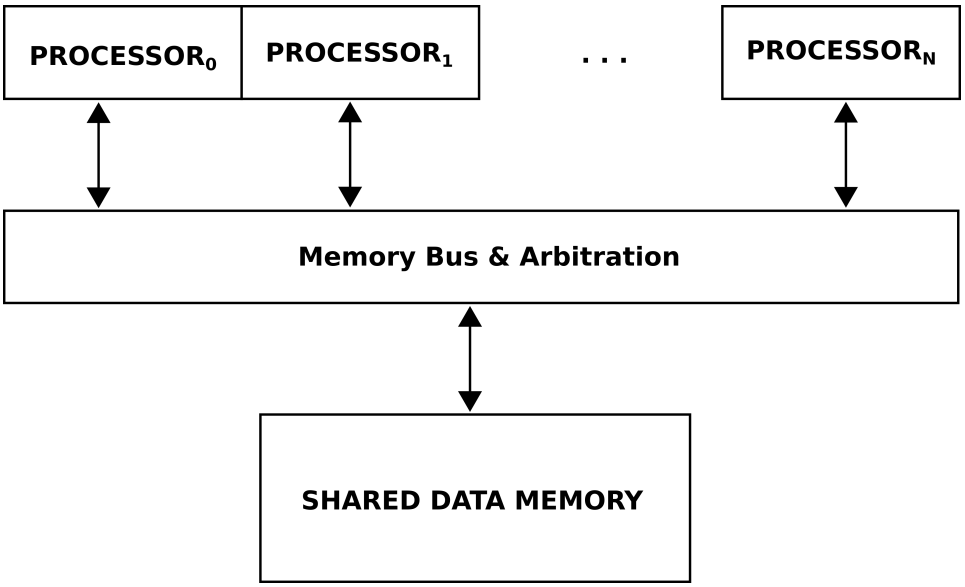
UMA

UMA je akronymem pro *Uniform Memory Access*. Žádný z procesorů není v přístupu do sdíleného paměťového prostoru zvýhodněn.

NUMA

NUMA je akronynem pro *Non-Uniform Memory Access*. Přístup do paměti je prioritizován a některý z procesorů je tak obsloužen dříve než jiný. Sdílený adresový prostor je typicky rozdělen do *banků* nebo *segmentů*, které pak přísluší jednotlivým procesorům.

[5] IPC > 1



Obrázek 1.4: Principiální schéma UMA zapojení

Počítač navržený v této práci lze tudíž klasifikovat jako minimalistický, *těsně vázaný* UMA víceprocesorový systém.

Poznámka. Na základě výše uvedených poznatků se zdržíme návrhu SIMD a pipeline a implementujeme *nezávislý* SISD výceprocesorový system. Klíčová rozhodnutí, která vedla k tomuto závěru jsou následující:

1. **Nedostatečná plocha čipu.** Implementace pipeline by vyústila ve veliké navýšení zdrojů chipu, které už jsou nyní skoro vyčerpány(7.2).
2. **Zpětná kompatibilita.** Navržený víceprocesorový systém se skládá ze čtyřech výkonných jednotek, přičemž návrh individuální procesorové jednotky byl předmětem bakalářské práce. SIMD představuje velkou změnu v celkové architektuře procesoru a porušila by mnoho definic učiněných dříve v bakalářské práci.
3. **Jednoduchost.** Celý systém byl navrhnut s ohledem na jednoduchost. Představení tolika architektonických změn by tudíž bylo zcela kontraproduktivní a vedlo k porušení tohoto kritéria.

Podrobnější detaily návrhu budou rozebrány v následujích kapitolách textu.

1.4 Reálná čísla

V tomto oddíle budou představeny základní metody reprezentace reálných čísel v digitalním počítači.

1.4.1 Plovoucí čárka

Čísla v plovoucí čárce jsou obecně reprezentována ve tvaru[1]

$$(-1)^S \times F \times B^E \tag{1.1}$$

kde S je znaménkový *sign* bit, F significand představující zlomkovou (*fractional*) část a E exponent o základu B. Existuje řada specifikací čísel v plovoucí čárce, mezi nimiž je dominantní zejména standard IEEE 754[11].

Příslušný standard musí pečlivě zvážit bitové šířky přiřazené zlomkové a exponenciální části, neboť je vždy nutné učinit kompromis mezi *přesností* a *rozsahem*. Rozšířením significandu F dosáhneme vyšší přesnosti, zatímco rozšířením exponentu E zvýšíme celkový rozsah numerických hodnot, které mohou být reprezentovány. Jelikož počet bitů je konečný, zvyšování přesnosti je možné pouze na úkor rozsahu a naopak.

Protože hranice mezi integrální a zlomkovou částí je *plovoucí*, je možné vyjádřit zároveň současně velmi malé a současně velmi velké hodnoty. Této vlastnosti říkáme dynamický rozsah. Z těchto důvodů se však k plovoucí reprezentaci váže několik fundamentálních problémů:

- Čísla nejsou *přesná* (exact) a skutečná reprezantace nuly je problematická, protože výsledek početní operace nebude nikdy v dané reprezentaci roven nule. Samotná nula (a tudíž i nekonečno) má rovněž znaménko ± 0, což může vést na spoustu neurčitých nebo nesprávných výrazů.
- Rozdíl dvou velmi malých, téměř shodných, čísel vyústí v naprosto chybný výsledek. Příčinou je, že nízké číslo má většinu bitů significandu F rovnou nule a zbývající bity jsou již pouze systematické chyby po častém zaokrouhlení. Ze stejného důvodu je i velmi těžké porovnat, zda jsou daná dvě čísla shodná.

Z výše uvedeného je patrné, že největším problémem čísel v reprezentaci plovoucí čárky je *numerická nestabilita*.

Formát čísla v reprezentaci plovoucí čárky je obdobou čísla zapsaného pomocí *vědecké notace* (scientific notation). Aritmetické operace vykonané na číslech v plovoucí čárce budou tudíž podléhat stejným principům, jako manipulace s čísly ve vědecké notaci.

Součet

Součet dvou operandů $(-1)^{S_1} \times F_1 \times B^{E_1} + (-1)^{S_2} \times F_2 \times B^{E_2}$ je proveden následovně:

1. Operandy součinu je nutno upravit tak, aby platilo

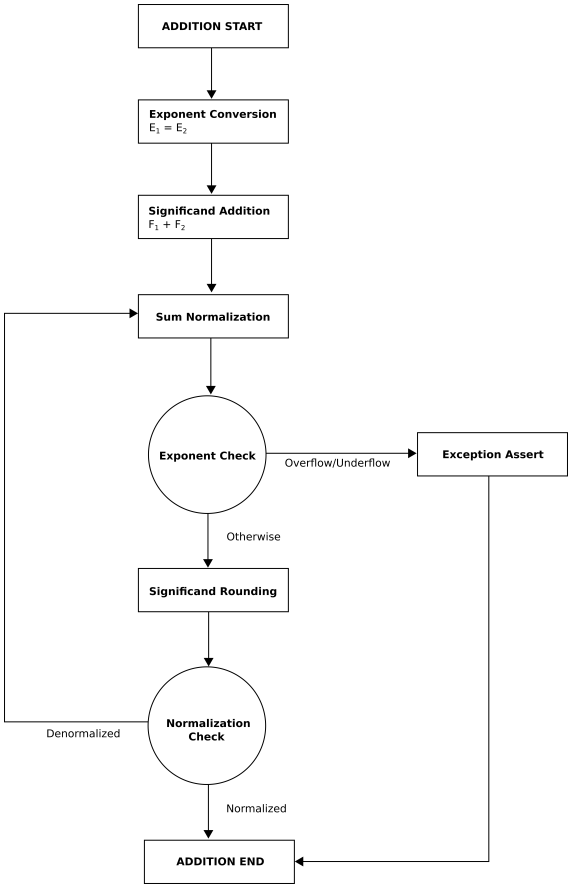
$$E_1 = E_2 \tag{1.2}$$

   Jakmile jsou si exponenty rovny, výsledek je dále zaokrouhlen na platný počet bitů significandu.

2. Significandy F_1 a F_2 mohou být nyní sečteny a výsledek patřičně zaokrouhlen.

3. Po dokončení kalkulace musí být zaručeno, aby výsledek zůstal normalizován.

Schematicky je tato procedura znázorněna níže:



Obrázek 1.5: Součet v notaci plovoucí čárky

Součin a podíl

Součin dvou operandů $(-1)^{S_1} \times F_1 \times B^{E_1} \cdot (-1)^{S_2} \times F_2 \times B^{E_2}$ je proveden následovně:

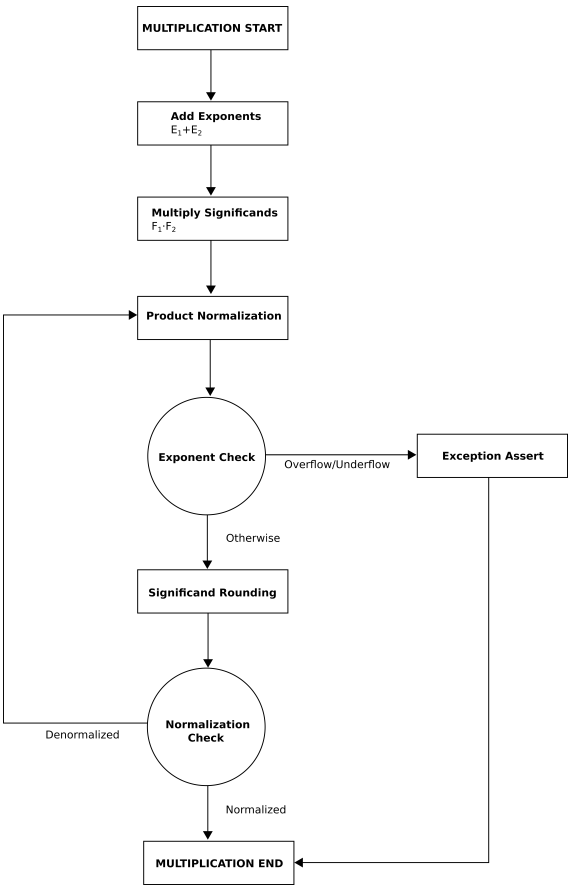
1. Exponenty E_1, E_2 jsou sečteny.

2. Significandy F_1 a F_2 mohou být nyní vynásobeny a výsledek patřičně zaokrouhlen.

3. Po dokončení kalkulace musí být zaručeno, aby výsledek zůstal normalizován.

4. Znaménkový bit S součinu je nastaven v závislosti na znaménkových bitech S_1, S_2 obou operandů. Pokud $S_1 = S_2$ pak $S = 0$, jinak je bit S nastaven $S = 1$. To odpovídá provedení operace $S = S_1 \oplus S_2$.

Podíl lze provést obdobně, namísto součtu však exponenty E_1, E_2 odečítáme a significandy F_1, F_2 vydělíme.



Obrázek 1.6: Součin v notaci plovoucí čárky

1.4.2 Pevná čárka

Čísla v pevné čárce (*fixed point*) vyhrazují přesný počet *číslic* d_i pro integrální a zlomkovou část čísla. Číslo o základu B má jednoduše tvar polynomu

$$d_n \cdot B^n + d_{n-1} \cdot B^{n-1} + \ldots + d_{-n} \cdot B^{-n} \tag{1.3}$$

kde d_i je i-tá číslice a n je celkový počet bitů čísla. Číslo v reprezentaci pevné čárky je ve skutečnosti celočíselné a čárka oddělující řády B^0, B^{-1} je implicitní. Zároveň je taková reprezentace reálného čísla člověku velmi přiřozená. Numerický typ reprezentovaný v pevné čárce má oproti plovoucí čárce řadu výhod, a to zejména:

- Rozsah je vždy konstatní a proto numerická hodnota v pevné čárce není pouze aproximací, ale má skutečnou hodnotu (*exact*). Rozestup mezi reprezentovatelnými reálnými čísly má uniformní rozložení a nulu lze vyjádřit přesně.
- Jelikož má dané číslo celočíselný formát, veškeré aritmetické operace lze provádět úplně standardním způsobem, až na drobné korekce měřítka v případě součinu a podílu. Tato vlastnost rovněž extrémně zjednodušuje nároky na další hardware.
- Formát čísel v pevné čárce lze velmi snadno přizpůsobit (a navrhnout) pro dedikované operace.
- Pokud je zvolen formát o základu dvě, změnu měřítka lze provést velmi jednoduše a rychle pomocí prostého bitového posunu.

Součet (rozdíl) dvou čísel v pevné čárce vyžaduje korekci pro rovnost měřítek operandů. Po korekci je možné operandy klasicky sečíst. Níže se tak zaměříme pouze na korekci součinu a podílu.

Součin

Nejprve provedeme klasický celočíselný součin, přičemž operandy *nemusí* být stejného formátu. Výsledný formát bude součtem počtu bitů integrální a zlomkové části. Součin je tudíž platný okamžitě, nicméně pokud počítač je navržen pro jediný formát, pak je nutná korekce pevné čárky. Protože nové měřítko je efektivně součinem jednotlivých měřítek operandů, stačí na výsledek aplikovat pravý bitový posun o počet bitů, který je roven absolutní hodnotě rozdílu bitové šířky zlomkových částí výsledku a cílového formátu. Tato operace vyžaduje zaokrouhlení nebo vynulování bitů (*truncate*).

Poznámka. Zde jsme implicitně předpokládali binární reprezentaci čísla. V případě decimální reprezentace je princip totožný, namísto bitového posunu je použit posun *desetinný*. Obecně je výsledek nutno vynásobit společným měřítkem.

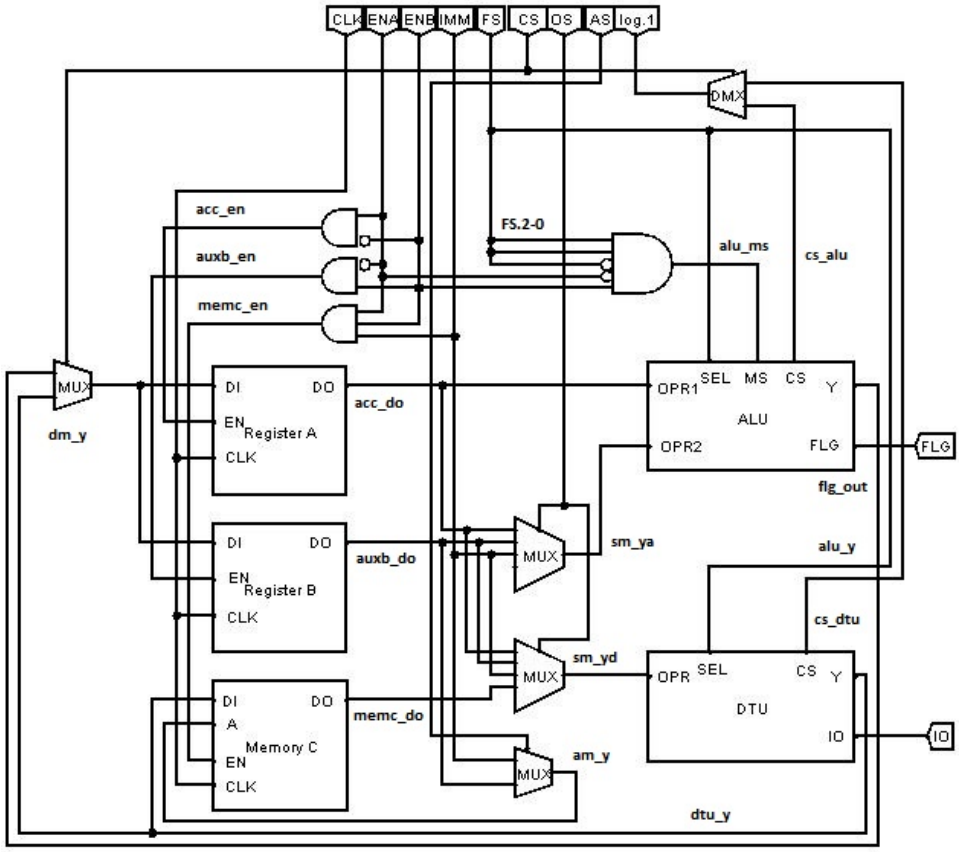
Podíl

Podíl je v principu velmi podobný součinu. Operandy opět nemusí být stejného formátu, nicméně některé kombinace vyústí v chybný výsledek. Výsledný formát bude nyní rozdílem počtu bitů integrální a zlomkové části. Tímto způsobem je například možné získat záporný výsledek při podílu dvou kladných čísel, neboť počet bitů nového formátu může být *záporný*, tzn. bity vyhrazené pro integrální část jsou přepsány zlomkovými bity a dojde tak k porušení integrity celé operace.

Můžeme si povšimnout, že podíl dvou operandů stejného formátu automaticky vyústí ve ztrátu přesnosti kvůli nedostatku platných číslic. Tento problém lze mitigovat rozšířením dělence o další platné číslice, tzn. aplikování levého bitového (případně destinného, pokud jsou operandy základu deset) posunu. Obdobně jako v případě součinu je pak výsledek nutno poopravit o společné měřítko, v tomto případě dělením. Společné měřítko je pak rovno podílu individuálních měřítek operandů.

1.5 Původní práce

Tento text přímo navazuje na bakalářskou práci[12] a to rozšířením jednoprocesové systému na víceprocesorový. Implementovaný návrh využívá původní MMIO, datovou a programovou paměť. Každá z výkonných jednotek počítače je samostatný procesor původně navržený v[12]. Z důvodu konzistence textu zde proto alespoň okrajově uvedeme nejdůležitější prvky návrhu, jako je schéma datové cesty s definicemi kontrolních signálů a přehled instrukční sady spolu s kódováním jednotlivých instrukcí.



Obrázek 1.7: Datová cesta výkonné jednotky

Označení	Plný Název
SIGNÁLY DATOVÉ CESTY	
dm_y	Destination Multiplexer Output
acc_do	Accumulator Data Output
auxb_do	Auxiliary Data Output
memc_do	Memory Data Output
sm_ya	ALU Source Multiplexer Output
sm_yd	DTU Source Multiplexer Output
acc_en	Enable Accumulator
memc_en	Enable Memory

Označení	Plný název
SIGNÁLY DATOVÉ CESTY	
am_y	Address Multiplexer Output
cs_alu	ALU Chip Select
ds_dtu	DTU Chip Select
alu_y	ALU Output
dtu_y	DTU Output
flg_out	Flags Output
aubx_en	Enable Auxiliary
alu_ms	ALU Multiply Select

Tabulka 1.1: Výčet interních signálů datové cesty

S_{HEX}	Název	Poznámka
MNEMONICKÉ OZNAČENÍ		
00	Fetch0	Výchozí stav
01	Fetch1	Dekódováni
02	ADD B	
03	ACC Write-Back	
04	ADD IMM	
05	NOT B	
06	AND B	
07	AND IMM	
08	OR B	
09	OR IMM	
0A	MOV IMM	
0B	MOV IMM	
0C	AUX Write-Back	
0D	MOV R	
0E	MOV R	
10	PIO IO	
11	B	Interní
12	BEQ	Interní
13	STO IMM	
14	LOD IMM	
15	SLL/CMS	CMS interní

S_{HEX}	Název	Poznámka
MNEMONICKÉ OZNAČENÍ		
16	SLR/CHHP/CLHP	
17	MPY	
18	DIV	
19	STO B	
1A	LOD B	
1B	CALL	Interní
1C	RET	Interní
1D	DAB	Sekvence
1E	Stack OFF	Interní
1F	Comp. Alarm	Interní
20	MPY Write-Back	MS = 0
21	MPY Write-Back	MS = 1
22	DAB Decrement	
23	DAB Write-Back	
24	DAB Branch	
25	RSVD	Rezerva
26	DAB Write-Back	
27	DAB Write-Back	
28	Read Stack	Interní
29	Wait	
2A	PIO Write-Back	

Tabulka 1.2: Přehled stavů kontrolní jednotky

<table>
<tr><td>CS.0</td><td>Chip Select. Prostřednictvím tohoto signálu vybírá demultiplexor ALU nebo DTU komponentu. Pokud CS = 0, komponenta ALU je aktivována, v opačném případě (CS = 1) je aktivována komponenta DTU. Pro aktivovanou komponentu se pak stává relavantní signál FS. Výstup demultiplexoru je vždy log.1 pro aktivovanou komponentu a log.0 pro deaktivovanou komponentu. Signál CS rovněž ovládá cílový multiplexor, pomocí kterého je veden zpětný zápis do registru.</td></tr>
<tr><td>OS.0–1</td><td>Operand Select. Tento signál ovládá multiplexor, který připojuje datové výstupy registru A,B, datové paměti a IMM na zdrojové vstupy komponent ALU a DTU. Možné hodnoty kterých může OS.0–1 nabývat jsou uvedeny v tabulce níže.

<table>
<tr><th>OS1</th><th>OS0/CS</th><th>Zdroj – ALU/DTU</th><th>Zdroj – A/B</th></tr>
<tr><td>0</td><td>0</td><td>Registr A</td><td>ALU</td></tr>
<tr><td>0</td><td>1</td><td>Registr B</td><td>DTU</td></tr>
<tr><td>1</td><td>0</td><td>Paměť</td><td>NC[6]</td></tr>
<tr><td>1</td><td>1</td><td>Immediate</td><td>NC</td></tr>
</table></td></tr>
<tr><td>FS.1–0</td><td>Function Select. Tento signál vybírá specifickou akci kterou by komponenta měla provést. Cílová komponenta musí mít CS v log.1, jinak je tento signál ignorován. Platné hodnoty, jakých může signál nabývat, jsou uvedeny v tabule níže pro každou z komponent zvlášť.

<table>
<tr><th>FS2</th><th>FS1</th><th>FS0</th><th>ALU Akce</th><th>DTU Akce</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>ADD</td><td>MOV</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>NOT</td><td>PIO IO</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>AND</td><td>PIO A</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>OR</td><td>CHHP</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>SLL</td><td>CHLP</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>SLR</td><td>Rezerva</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>MPY</td><td>Rezerva</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>DIV</td><td>Rezerva</td></tr>
</table></td></tr>
<tr><td>AS.0</td><td>Address Select. Tento signál ovládá adresový multiplexor, jehož výstup je směrován na adresový port A datové paměti. Multiplexor tak vybírá mezi adresovými vstupy, kterými mohou být registry okamžité hodnoty IMM (AS = 0) nebo registr B (AS = 1).</td></tr>
<tr><td>IMM.3–0</td><td>Immediate. Obsahuje výstup z registru okamžité hodnoty IMM, který je přiveden na vstup multiplexoru pro komponenty ALU a DTU. Pokud OS = 3, objeví se tato hodnota IMM registru na OPR vstupech obou komponent. IMM není kontrolní signál.</td></tr>
</table>

<table>
<tr><td>ENA.0</td><td>Enable Accumulator. Ovládá zápis do registru A. Pokud ENA = 1 obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna.</td></tr>
<tr><td>ENB.0</td><td>Enable Auxiliary B. Ovládá zápis do registru B. Pokud ENB = 1 obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna.</td></tr>
<tr><td>END.0</td><td>Enable IO. Hlavní EN signál, který ovládá individuální EN signály vstupních, výstupních a konfiguračních IO registrů. Alias signálu je ENIO. Kontrolované signály jsou shrnuty v tabulce níže.

<table>
<tr><th>END</th><th>FS</th><th>Funkce</th><th>Signal/Reg</th><th>Instrukce</th></tr>
<tr><td>1</td><td>1</td><td>ENOUT</td><td>IOUPT</td><td>PIO IO</td></tr>
<tr><td>1</td><td>2</td><td>ENIN</td><td>IOINPT</td><td>PIO A</td></tr>
<tr><td>1</td><td>3</td><td>ENHHP</td><td>IOCFG</td><td>CHHP</td></tr>
<tr><td>1</td><td>4</td><td>ENLHP</td><td>IOCFG</td><td>CLHP</td></tr>
</table>
<table>
<tr><td>ENOUT</td><td>Enable Output.</td><td>ENOUT $\models\!\models$ END $\wedge \neg$FS.2 $\wedge \neg$FS.1 $\wedge$ FS.0</td></tr>
<tr><td>ENIN</td><td>Enable Input.</td><td>ENOUT $\models\!\models$ END $\wedge \neg$FS.2 $\wedge$ FS.1 $\wedge \neg$FS.0</td></tr>
<tr><td>ENHHP</td><td>Enable High Half Port.</td><td>ENOUT $\models\!\models$ END $\wedge \neg$FS.2 $\wedge$ FS.1 $\wedge$ FS.0</td></tr>
<tr><td>ENLHP</td><td>Enable Low Half Port.</td><td>ENOUT $\models\!\models$ END $\wedge$ FS.2 $\wedge \neg$FS.1 $\wedge \neg$FS.0</td></tr>
</table>
Poznámka. Pravdivostní tabulka je zde zapsána velmi úsporně. Signál (funkce) nabývá hodnoty log.1 pouze pro uvedené hodnoty END a FS. Jinde je funkce nulová.</td></tr>
<tr><td>MS.0</td><td>Multiply Select. Pro hodnoty MS = 0 bude zapsáno spodních 12 bitů a pro MS = 1 horních 12 bitů ALU výstupu. Používáno MPY instrukcí. MS není kontrolní signál.</td></tr>
</table>

Tabulka 1.3: Přehled kontrolních signálů

Níže je uveden výpis všech instrukcí procesoru[7] (nyní individuální výkonné jednotky). Extra kódy XTR zde nejsou uvedeny, neboť jsou definovány až v textu dále.

[7] CE (*Condition Enable*) – možnost podmíněného vykonávání instrukce, FA (*Flags Affected*)– daná instrukce ovlivňuje stavové vlajky FLG registru.

Mnemonic	Operand	Description	CE	FA
ARITHMETIC AND LOGICAL OPERATIONS				
ADD	Aux B	Add registers.	Yes	Yes
ADD	IMM	Add immediate.	No	Yes
NOT	Aux B (implicit)	Negate Aux B.	Yes	Yes
AND	Aux B	Bitwise AND registers.	Yes	Yes
AND	IMM	Bitwise AND immediate.	No	Yes
OR	Aux B	Bitwise OR registers.	No	Yes
OR	IMM	Bitwise OR immediate.	No	Yes
DATA TRANSFER OPERATIONS				
MOV A	IMM	Move immediate.	No	No
MOV B	IMM	Move immediate.	No	No
MOV A	Aux B (implicit)	Move registers.	Yes	No
MOV B	A (implicit)	Move registers.	Yes	No
LOD	IMM	Load Accumulator.	No	No
STO	IMM	Store Accumulator.	No	No
INPUT/OUTPUT OPERATIONS				
PIO	IO	Process Input and Output.	Yes	No
PIO	A	Process Input and Output.	Yes	No
TRANSFER CONTROL OPERATIONS				
B	IMM	Branch to immediate.	No	No
BEQ	IMM	Branch to immediate if A = 0.	No	No
Mnemonic	Operand	Description	CE	FA[8]
ARITHMETIC AND LOGICAL OPERATIONS – EXTEND				
SLL	IMM	Compare registers.	Yes	Yes
SLR	IMM	Load indirect.	Yes	Yes
MPY	AB (implicit)	Multiply registers.	Yes	Yes
DIV	AB (implicit)	Divide registers.	Yes	Yes
DATA TRANSFER OPERATIONS – EXTEND				
STO	Aux B	Store indirect.	Yes	No
LOD	Aux B	Load indirect.	Yes	No
INPUT/OUTPUT OPERATIONS – EXTEND				
CHHP	IMM	Change high half port.	No	No
CLHP	IMM	Change low half port.	No	No
TRANSFER CONTROL OPERATIONS – EXTEND				
CALL	IMM	Call procedure.	No	No
RET	None	Return from procedure.	Yes	No
DAB	IMM (IMM)	Decrement A and branch.	No	Yes
CMS	IMM	Change memory segment.	No	No

Tabulka 1.4: Přehled instrukcí výkonné jednotky

ADD	B	X	X	X	X	X	X	X	X	Z	Y	0	0	0	0	0	0
ADD	IMM	D	D	D	D	D	D	D	D	D	D	D	D	1	0	0	0
NOT	B	D	D	D	D	D	D	D	D	D	D	D	D	1	0	0	0
AND	B	X	X	X	X	X	X	X	X	X	Z	Y	0	0	0	1	0
AND	IMM	D	D	D	D	D	D	D	D	D	D	D	D	1	0	1	0
OR	B	X	X	X	X	X	X	X	X	X	Z	Y	0	0	0	1	1
OR	IMM	D	D	D	D	D	D	D	D	D	D	D	D	1	0	1	1
MOV A	IMM	D	D	D	D	D	D	D	D	D	D	D	D	0	1	0	0
MOV B	IMM	D	D	D	D	D	D	D	D	D	D	D	D	1	1	0	0
MOV A	B	X	X	X	X	X	X	X	X	X	Z	Y	0	0	1	0	1
MOV B	A	X	X	X	X	X	X	X	X	X	Z	Y	0	1	1	0	1
LOD	IMM	0	A	A	A	A	A	A	A	A	A	A	A	0	0	0	1
STO	IMM	1	A	A	A	A	A	A	A	A	A	A	A	0	0	0	1
PIO	IO	0	X	X	X	X	X	X	X	X	Z	Y	0	0	1	1	0
PIO	A	X	X	X	X	X	X	X	X	X	Z	Y	0	1	1	1	0
B	IMM	A	A	A	A	A	A	A	A	A	A	A	A	0	1	1	1
BEQ	IMM	A	A	A	A	A	A	A	A	A	A	A	A	1	1	1	1
EXTEND																	
SLL	IMM	0	X	X	X	D	D	D	D	X	Z	Y	1	0	0	0	0
SLR	IMM	1	X	X	X	D	D	D	D	X	Z	Y	1	0	0	0	0
MPY	AB	0	X	X	X	X	X	X	X	X	Z	Y	1	0	0	1	0
DIV	AB	1	X	X	X	X	X	X	X	X	Z	Y	1	0	0	1	0
STO	B	1	X	X	X	X	X	X	X	X	Z	Y	1	0	1	0	1
LOD	B	0	X	X	X	X	X	X	X	X	Z	Y	1	0	1	0	1
CHHP	IMM	1	0	D	D	D	D	D	D	X	1	1	1	0	0	0	0
CLHP	IMM	1	1	D	D	D	D	D	D	X	1	1	1	0	0	0	0
CALL	IMM	0	A	A	A	A	A	A	A	A	A	A	1	0	0	1	1
RET	–	1	X	X	X	X	X	X	X	X	Z	Y	1	0	0	1	1
DAB	IMM	A	A	A	A	A	A	A	A	B	B	W	1	0	1	1	0
CMS	IMM	0	X	X	D	D	D	D	D	D	1	1	1	0	0	0	0

Tabulka 1.5: Přehled kódování instrukcí výkonné jednotky

Poznámka. Podrobné informace o návrhu individální výkonné jednotky (procesoru) lze nalézt v původní práci[12].

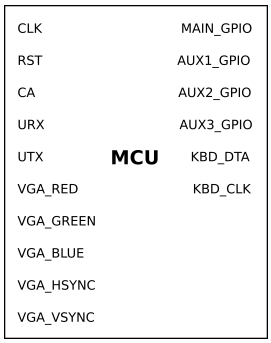
Kapitola 2

Návrh mikropočítače

V této kapitole bude rozebrán vlastní návrh počítače. Hlavní charakteristiky počítače lze shrnout do několika málo bodů:

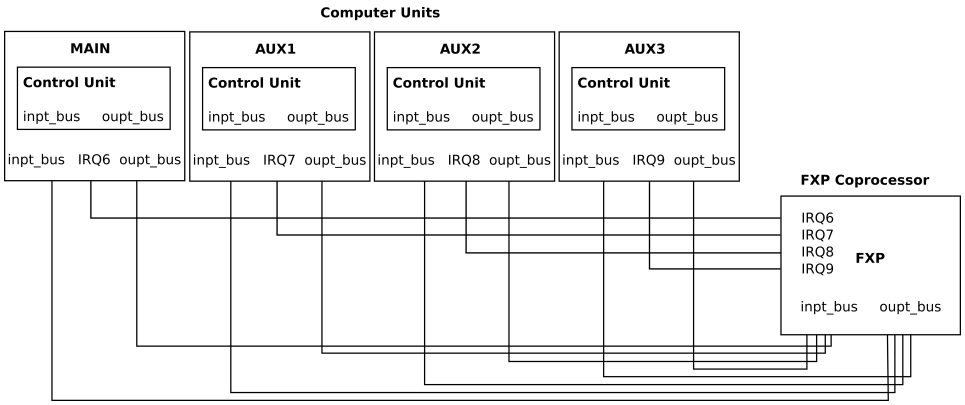
- Celkem čtyři nezávislé výpočetní jednotky.
- 64k slov sdílené programové paměti rozdělené do 32 segmentů
- Architektura může podporovat až 256k slov sdílené datové paměti rozdělené do 128 segmentů.
- Matematický koprocesor FXP.
- Integrovaný VGA video ovladač a PS/2 kompatibilní ovladač.

Následuje schématická značka implementovaného počítače spolu s popisem všech pinů.



Obrázek 2.1: Top module schematic symbol

CLK	MCU.00	IN	**Clock.** Hlavní hodinový signál 100 MHz.
RST	MCU.01	IN	**Reset.** Reset počítače.
CA	MCU.02	OUT	**Computer Alarm.** Počítač je v bezpečném režimu.
URX	MCU.03	IN	**Receive.** UART Rx vodič.
UTX	MCU.04	OUT	**Transmit.** UART Tx vodič.
VGA_RED	MCU.05-07	OUT	**Red.** Červená komponenta VGA video signálu.
VGA_GREEN	MCU.08-10	OUT	**Green.** Zelená komponenta VGA video signálu.
VGA_BLUE	MCU.11-12	OUT	**Blue.** Modrá komponenta VGA video signálu.
VGA_HSYNC	MCU.13	OUT	**Horizontal Sync.** Horizontální video synchronizace.
VGA_VSYNC	MCU.14	OUT	**Vertical Sync.** Vertikální video synchronizace.
MAIN_GPIO	MCU.15-26	I/O	**Input and Output.** Unit 0 general purpose input and output.
AUX1_GPIO	MCU.27-38	I/O	**Input and Output.** Unit 1 general purpose input and output.
AUX2_GPIO	MCU.39-50	I/O	**Input and Output.** Unit 2 general purpose input and output.
AUX3_GPIO	MCU.51-62	I/O	**Input and Output.** Unit 3 general purpose input and output.
KBD_DTA	MCU.63	I/O	**Keyboard Data.** Datový signál PS/2 portu.
KBD_CLK	MCU.64	I/O	**Keyobard Clock.** Hodinový signál PS/2 portu.



Obrázek 2.2: Propojení FXP koprocesoru a počítače

Ačkoliv je návrh kontrolní jednotky zpětně zcela kompatibilní s původní prací[12], bylo pro účely arbitrace datové a programové paměti nutné implementovat některé další stavy. Interprocesová komunikace mezi počítačem a FXP koprocesorem si rovněž vyžádala rozšíření stavů kontrolní jednotky.

Z úsporných důvodů v tomto textu uvedene pouze stručný přehled nově implementovaných stavů.

S_{HEX}	Název	Poznámka	S_{HEX}	Název	Poznámka
MNEMONICKÉ OZNAČENÍ			**MNEMONICKÉ OZNAČENÍ**		
2B	LOD IMM GRANT		40	Context Restore	
2C	STO IMM GRANT		41	ACC Write-Back	
2D	LOD B GRANT		42	ST Address Load	
2E	STO B GRANT		43	DIV WAIT	DIV sekv.
2F	PROG GRANT		44	RETI Stack OFF	
30	IDLE	Pouze AUX	45	RETI LOAD PC	
31	LOD PROG GRANT		46	LD IMM	
32	XCHG Write-Back	Registr B	47	LD GRANT	
33	XCHG Write-Back	Registr A	48	LD Write-Back	Sekvence
34	PROG Write-Back		49	LD Address Inc.	
35	Wait Program	Req. assert	4A	ST START	Bus transfer
36	FXP WAIT ACK	Bus transfer	4B	SCSW GRANT	XTR transfer
37	FXP WAIT STA	Bus transfer	4C	SCSW LOAD	Bus transfer
38	ST Write-Back	Sekvence	4D	LCSW Address Inc.	
39	ST GRANT		4E	LIW	IFR load
3A	ST Address Inc.		4F	SIW	AB select
3B	ST Write-Back	Slovo 1	50	SIW Write-Back	
3C	ST GRANT		X	(Prázdné)	
3D	ST Write-Back	Slovo 2	X	(Prázdné)	
3E	ST Write-Back	Slovo 3	X	(Prázdné)	
3F	ST Address Inc.		X	(Prázdné)	

Tabulka 2.1: Přehled stavů kontrolní jednotky

2.1 Auxiliární jednotky

Implementovaný počítač obsahuje celkem čtyři samostatné výpočetní jednotky: MAIN, AUX1, AUX2 a AUX3. Výpočetní jednotky jsou zcela identické s tím rozdílem, že MAIN nelze vypnout. Jednotky AUX1, AUX2 a AUX3 jsou vedeny jako *auxiliární* a mohou být kdykoliv suspendovány či spuštěny nastavením přislušných bitů v **Computer Control Register** (CCR). Počáteční adresu PC dané jednotky je před startem nutno nakonfigurovat v odpovídajícím **Load Registru** (LR). Registry LR jsou *stínovany* (shadowed) – každá z výkonných jednotek počítače tak má svůj patřičný registr.

CCR : COMPUTER CONTROL REGISTER **Write: 0x7F7** **Read: 0x7F7**

MSB											**LSB**
RSV	RSV	RSV	RSV	RSV	RSV	IDL	IDL	IDL	STA	STA	STA

STA	CCR.00	R/W	**Start.** Pokud je tento bit nastaven a výkonná jednotka je v IDLE stavu, pak bude programový čítač PC nastaven na adresu specifikovanou v registru LR.
IDL	DCR.01	R/W	**Idle.** Pokud je tento bit nastaven, příslušná jednotka je suspendována, vstoupí do IDLE stavu a veškeré příkazy budou pozastaveny až do opětovného resetování IDLE bitu.
RSV	DCR.03-11	NI	**Reserved.** Čteno jako nula.

Pro zapnutí zvolené auxiliární jednotky musí prográm nejprve provést následující sekvenci:

1. Nastavení IDL (Idle) bitu v CCR. Jednotka vstoupí do IDLE stavu.
2. Nahrání počáteční adresy do load registru LR. Toto je dvoufázový proces, neboť programový čítač je 16-bitů široký.
3. Nastavení STA (Start) bitu v CCR.
4. Resetování STA a IDL bitů v CCR. Jednotka vstoupí do FETCH stavu a je nyní zapnuta.

Pro vypnutí jednotky stačí nastavit IDL bit. Jednotka vstoupí do IDLE stavu, přičemž zachová veškerý programový kontext, tzn. PC, A, B, IO registry a FLG. Nastavení STA bitu, zatímco je IDL bit resetován, nemá žádný efekt.

Níže je uveden příklad uživatelského kódu, který implementuje proceduru (2.1).

```
; Turn on Auxiliary Unit #1
    lod     #CCR
    or      #8      ; Set the IDL bit
    and     #0xFFE  ; Clear STA bit
    sto     #CCR
    mov     b, a
    mov     a, .unit1_entry'high
    sto     #LR1_H
    mov     a, .unit1_entry'low
    sto     #LR_L
    mov     a, b
    or      #1      ; Set STA bit
```

```
    sto     #CCR
    and     #0xFF6 ; Clear ILD and STA bits
    sto     #CCR
; AUX1 is ON now
...
.unit1_entry:
...                 ; AUX1 User Code
```

- Po přivedení napájení a zapnutí počítače jsou všechny STA bity resetovány a všechny IDL bity nastaveny. Všechny auxiliární jednotky jsou tudíž ve výchozím stavu vypnuty.

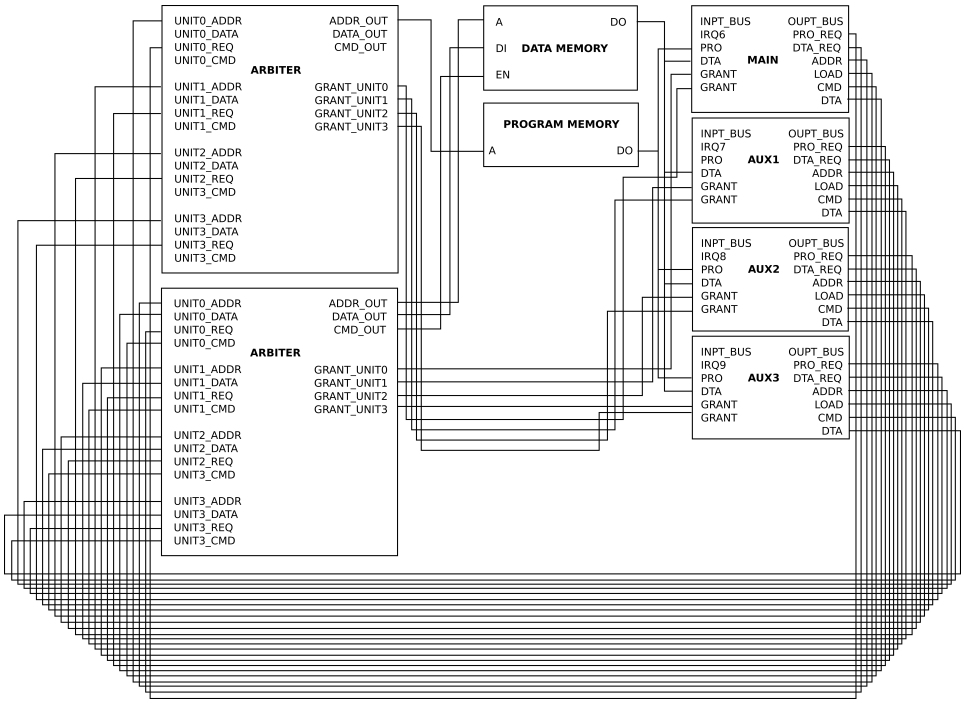
- Pokud je auxiliární jednotka v provozu a je asertován resetovací pin RST počítače, jednotka automaticky nastaví programový čítač PC na současnou hodnotu v load registru LR. Jakmile je příslušný IDL bit resetován, jednotka je opět zapnuta.

2.1.1 Arbiter

Výkonné jednotky počítače spolu sdílí programový a datový prostor. Přístup do programové a datové paměti tudíž musí být vhodným způsobem plánován, multiplexován a *arbitrován*. Vstupy datové paměti jsou adresový, datový a *write-enable* vodič, zatímco výstupem je pak (*registrovaný*) vodič datový. Obdobně vstupem a výstupem paměti jsou registrované adresové a datové vodiče, respektivně. Dle návrhu jsou obě paměti *jednoportové* (single-port), nicméně každá z výkonných jednotek má svoji vlastní, nezávislou sadu vstupních a výstupních vodičů. Tyto vstupy a výstupy musejí být přepínány tak, aby v jednu chvíli byla uznána žádost pouze jedné jediné jednotky a bylo tak možné zachovat tradiční princip jednoportové paměti.

Poznámka. Alternativně bychom mohli problém arbitrace řešit konstrukcí čtyřportových pamětí. Toto řešení je však zhlediska časování a využité plochy chipu nevhodné. Z hlediska návrhových pravidel je doporučováno dodržovat princip arbitrované jednoportové paměti, který je pro syntetizační nástroj snadněji uchopitelný. Víceportové paměti jsou v návrhu implementovány např. v subsystému video ovladače, kterému je věnována kapitola (4.1).

Níže je znázorněno schéma zapojení arbitrované programové a datové paměti.

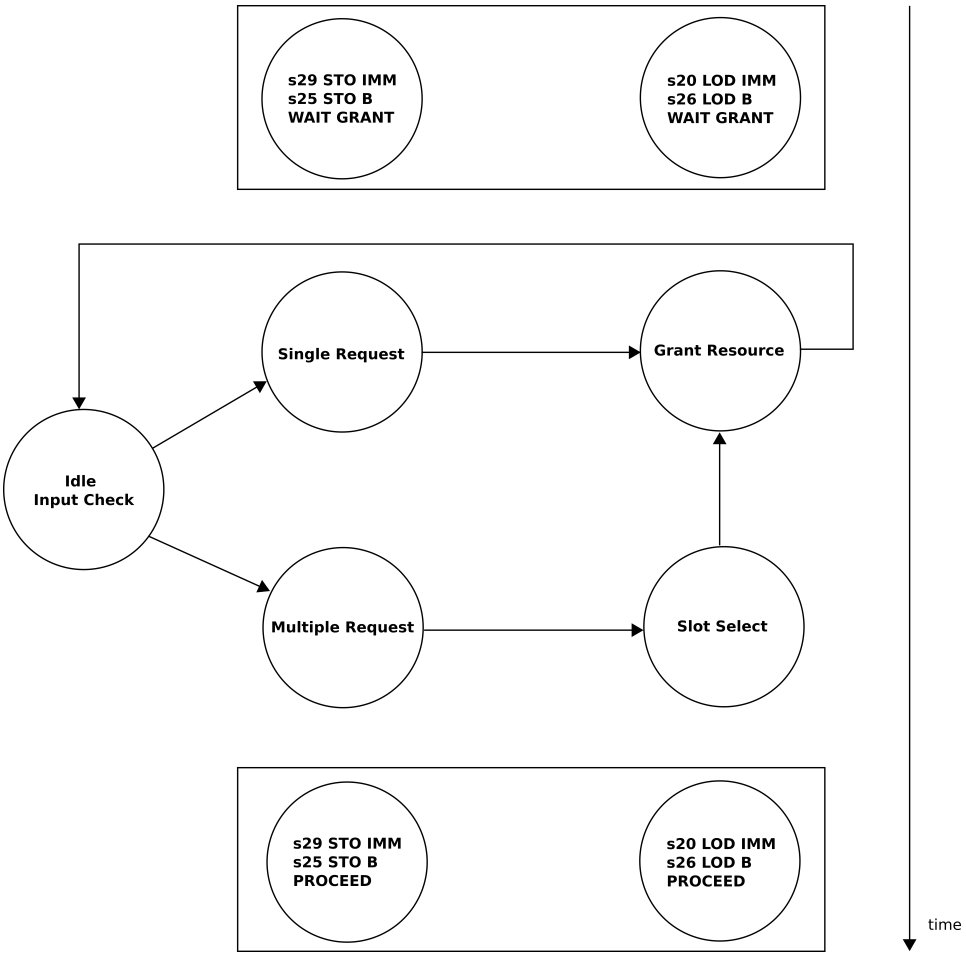


Obrázek 2.3: Zapojení sdílené programové a datové paměti

Jelikož žádná z výkonných jednotek neimplementuje cache a celý návrh je od počátku pro účely jednoduchosti stavěn okolo dvou *pracovních* registrů[1], utilizace datové paměti je velmi vysoká. Pro korektní fungování víceprocesového systému tudíž plánovač arbitrace musí zaručit spolehlivý a *spravedlivý* přístup do paměti a zabránit tak jevu tzv. *resource starvation*. Situace, kdy je jedné nebo vícero jednotkám kontinuálně odpírán přístup k požadovaným zdrojům – přístup do paměti, výpočetní čas, IO, síťové adaptéry ; reálně tedy libovolné sdílené zařízení či veličina. K tomuto jevu dochází převážně v prioritizovaných systémech, kde objekt vyšší priority může cyklicky odebírat zdroje na úkor objektů priority nižší. Tento návrh může rovněž vést i k paradoxním situacím, kdy zamítaný objekt nízké priority ohrozí činnost objektu vysoké priority. Je tedy patrné, že přerozdělování zdrojů má zcela zásadní a fundamentální vliv na funkčnost a stabilitu celého víceprocesorového systému.

Na diagramu níže je znázorněn implementovaný plánovací algoritmus.

[1] Podobně jako je tomu u řady jiných mikroprocesorů, jmenujme hlavně dominantní rodinu PIC procesorů.



Obrázek 2.4: Plánovací algoritmus arbiteru

S přihlédnutím k faktu, že každá z výkonných jednotek počítače je rovnocenná spolu s nároky na paměťovou utilizaci byl zvolen *round-robin* plánovač. Algoritmus není prioritizovaný, nezvýhodňuje žádnou z jednotek a rozděluje prostředky rovným dílem. Plánovač je konečný stavový automat, který vyčkává ve výchozím stavu do doby, kdy je alespoň jeden z *request* signálu aktivní. Interní modulo–4 čítač zde má funkci paměťového prvku a je inkrementován po každém vyhovění požadavku přístupu do paměti. Plánovač nastaví *grant* signál po dobu jednoho hodinového cyklu do aktivní pozice, inkrementuje čítač a takto pokračuje dokud nejsou obslouženy všechny zdrojové požadavky. Round-robin princip zaručuje, že žádná z výkonných jednotek systému nebude nepředvídatelně dlouho čekat na uznání příslušných zdrojů.

Níže je uveden postup procedury (2.4).

1. Daná jednotka nastaví všechny příslušné kontrolní signály pro přístup do datové nebo programové paměti spolu s *request* signálem a přejde do *Wait Grant* stavu.
2. Jednotka setrvá v *Wait Grant* stavu až do přijetí *grant* signálu. Jelikož je plánovač bezprioritní, maximální doba čekání je 12 hodinových cyklů v případě třech plnících se požadavků.
3. Jednotka resetuje *grant* signál a dále pokračuje v instrukčním cyklu.

Poznámka. V případě, že jsou auxiliární jednotky AUX1, AUX2 a AUX3 vypnuty[2] je *round-robin* princip potlačen a požadavek je arbitrem automaticky vyřízen ihned.

2.2 Ovladač přerušení

Interupce, neboli *přerušení*, je žádost vyslaná daným zdrojem signalizující okamžitou pozornost výkonné jednotky, či procesoru obecně. Přerušení patří mezi základní nástroje prioritizovaného systému. Událost vyšší priority přeruší vykonávání uživatelského programu o prioritě nižší. Procesor pak pozastaví výkon současného kódu, uloží programový kontext a provede skok na funkci obsluhy přerušení (ISR, Interrupt Service Routine). Po návratu z ISR procesor znovu automaticky obvnoví programový kontext a pokračuje v běhu původně pozastaveného programu.

Obecně rozlišujeme dva druhy přerušení:

- **Hardwarové přerušení.** Zdrojem IRQ je fyzické zařízení, např. pevný disk, síťové rozhraní nebo externí periférie. Hardwarové přerušení čistě *asynchronní* a může nastat během libovolné fáze instrukčního cyklu.
- **Softwarové přerušení.** Zdrojem IRQ je dedikovaná *trap* instrukce[3] či jiná speciální událost systému, např. *výjimka* (exception). Softwarové přerušení je *synchronní* a může nastat pouze v přesně dané fázi instrukčního cyklu.

Počítač implementuje jak hardwarové tak softwarové přerušení. Jednotlivé zdroje přerušení jsou maskovatelné v registrech **Interrupt Flag Register** (IFR) a **Interrupt Enable Register** (IER). *Asynchronní* přerušení je neprve nutné vhodně synchronizovat.

[2]Komponenta arbitru nemá žádnou informaci o stavu jednotek. Round-robin princip je zkrátka potlačen, kdykoliv je registrován pouze jeden jediný požadavek, což je nejpravděpodobnější právě když jsou ostatní jednotky suspendovány.

[3]Trap instrukce jsou typicky užívány pro *krokování* běžícího programu či pro systémová volání. Uživatelský kód s nižším stupněm oprávnění tak může zavolat kód s vyšším stupněm oprávnění – např. kernel funkce operačního systému.

1. Zdroj přerušení je na začátku každého hodinového cyklu registrován.
2. Ve Wait[4] stavu S_{41} kontrolní logiky výkonné jednotky je obnoven registr IFR.
3. Ve Fetch0[5] stavu S_0 je rozhodnuto, zda proběhne vektorování do ISR nebo bude přečteno programové slovo na současné adrese PC. Principiální signál, který takto vygeneruje patřičné IRQ je v návrhu označen jako **Master IRQ**.

Signál Master IRQ q je generován následující kombinační logikou

$$q \models\models e_0 \wedge (e_n \wedge m_{n-1}) \vee \ldots \vee (e_1 \wedge m_0) \tag{2.1}$$

kde e_i je i-tý *enable* bit v IER registru, m_i je i-tý *maskovací* bit v IFR registru a $n = 5$ je maximální počet zdrojů přerušení navýšený o jedničku.

Podrobnosti o signálech a registrech ovladače přerušení lze nalézt dále v podkapitole (2.2.3).

2.2.1 Interrupt Vector Table

Jednotlivé vektory přerušení jsou pevné a začínají na PC = 1. První instrukci v programové paměti PC = 0 tedy lze využít na nepodmíněný skok přes celou tabulku vekorů IVT na začátek uživatelského programu.

```
b .start     ; Skip IVT
b .main_irq  ; Main IRQ Vector
b .aux1_irq  ; Aux1 IRQ Vector
b .aux2_irq  ; Aux2 IRQ Vector
b .aux3_irq  ; Aux3 IRQ Vector
.start:      ; User code
```

Všechna přerušení jsou po resetu vypnuta. Počítač implementuje celkem 4 vektory přerušení, každé pro jednu z výkonných jednotek:

Vector	Purpose
PC 0001	dedicated to MAIN IRQ line
PC 0002	dedicated to AUX1 IRQ line
PC 0003	dedicated to AUX2 IRQ line
PC 0004	dedicated to AUX3 IRQ line

Tabulka 2.3: Tabulka přerušení IVT

[4] Výkonná jednotka čeká na vyčtení data z programové paměti.
[5] Výchozí stav.

Jakmile je vygenerováno IRQ, programový čítač PC je nastaven na adresu patřičného vektoru dle (2.2). Systém automaticky uloží programový kontext, tj. stavový registr FLG, programový segment CPS a návratovou adresu PC a rovněž tyto obnoví v případě návratu z ISR. Pracovní registry A a B musí být uloženy a obnoveny uživatelským softwarem manuálně.

Každý z vektorů typicky obsahuje větvící instrukci (B IMM) či pouze RETI, které efektivně přerušení ignoruje. Zde je však nutno podotknout, že zdroje přerušení je nutno resetovat ručně – v opačném případě bude dané IRQ voláno nepřetržitě až do vypršení zdroje.

2.2.2 Interrupt Service Routine

Obecně může systém registrovat všechna IRQ ve stejném čase, stejně tak jako mohou výkonné jednotky simultáně vektorovat do svých ISR. ISR (Interrupt Service Routine) je uživatelský software, jehož význam spočívá v obsloužení daného přerušení a to následujícím způsobem:

1. Uloží A, B do datové paměti.
2. Přečte Interrupt Flag Register (IFR) a zjistí zdroj současného IRQ.
3. Obslouží zdroj(e) IRQ.
4. Obnoví A, B.
5. Vykoná návrat z přerušení. Návrat z přerušení by měl být doprovázen resetováním patřičného zdrojového bitu v IFR. V opačném případě dojde znovu k vektoraci do ISR.

Uživatelský ISR software by měl v každém případě začínat uložením registrů A a B a končit jejich obnovením před návratem do hlavního programu, viz následující příklad.

```
sto #UNITx_ACC   ; store A
mov b, a
sto #UNITx_B     ; store B
...              ; actual ISR
lod #UNITx_B
mov b, a         ; restore B
lod #UNITx_A     ; restore A
reti             ; return control
```

Poznámka. Po dobu výkonu ISR jsou ostatní zdroje IRQ pozastaveny. Uživatelský software se tudíž nemusí zabývat případy, kdy by mohl být sám přerušen. I zde však obecně platí, že ISR kód by měl být pokud možno co nejkratší – ať už pouze z důvodu vypršení jednotlivých zdrojů IRQ (např. opětovný stisk klávesy či dokončení následující z iterativních FXP instrukcí.).

2.2.3 Registry přerušení

K přímé manipulaci s ovladačem přerušení jsou vyhrazeny dva registry, a to IER, který obsahuje jednotlivé *enable* bity zdrojů přerušení a IFR obsahující příslušné *vlajky* zdrojů přerušení. Registry IER a IFR jsou *zrcadleny* (shadowed) – každá z výkonných jednotek počítače implementuje vlastní kopii registrů IER a IFR. Přerušení lze povolit odmaskováním nadřazeného IER.MAST bitu a dále vybraných zdrojů přerušení. Po vygenerování hlavního přerušení dle vztahu (2.1) lze individuální zdroj určit *pollováním* vlajkového IFR registru. Registry lze číst a zapisovat instrukcemi LIW a SIW, respektivně. Na konci ISR je nutné dané vlajky zdrojů registru IFR resetovat, aby nedošlo o cyklickému vektorování již obslouženého přerušení.

Níže je uveden patřičný význam bitových polí registrů přerušení IER a IFR.

IER : INTERRUPT ENABLE REGISTER **Write: SIW** **Read: LIW**

MSB											**LSB**
RSV	RSV	RSV	RSV	RSV	RSV	EXT	STCK	TIM	KBD	FXP	MAST

RSV	IER.06-11	NI	**Reserved.** Čteno a zapisováno jako nula.
EXT	IER.05	R/W	**External.** Zapne či vypne zdroj externího přerušení.
STCK	IER.04	R/W	**Stack.** Zapne či vypne zdroj přerušení přetečení zásobníku SP.
TIM	IER.03	R/W	**Timer.** Zapne či vypne zdroj přerušení čítače. Pokud jsou Compare a Load registry shodné, potom STS.DON je zdrojem přerušení.
KBD	IER.02	R/W	**Keyboard.** Zapne či vypne zdroje přerušení klávesnice.
FXP	IER.01	R/W	**Coprocessor.** Zapne či vypne zdroje přerušení FXP koprocesoru. Více o zdrojech FXP přerušení lze nalézt v oddíle (3.1).
MAST	IER.00	R/W	**Master Enable.** Hlavní vypínač. Nastavení MAST = 0 efektivně ignoruje veškeré individuální zdroje přerušení a kompletně vypne celý subsystém pro příslušnou výkonnou jednotku.

Tabulka 2.4: Interrupt Enable Register

IFR : INTERRUPT FLAG REGISTER **Write: SIW** **Read: LIW**

MSB											**LSB**
RSV	RSV	RSV	RSV	RSV	RSV	EXT	STCK	TIM	KBD	FXP	MAST

RSV	IER.05-11	NI	**Reserved.** Čteno a zapisováno jako nula.
EXT	IER.04	R/W	**External.** Vlajka zdroje externího přerušení.
STCK	IER.03	R/W	**Stack.** Vlajka zdroje přerušení přetečení zásobníku SP.
TIM	IER.02	R/W	**Timer.** Vlajka zdroje přerušení čítače.
KBD	IER.01	R/W	**Keyboard.** Vlajka přerušení klávesnice.
FXP	IER.00	R/W	**Coprocessor.** Vlajka zdroje přerušení FXP koprocesoru.

Tabulka 2.5: Interrupt Flag Register

2.3 Matematický koprocesor

Řada aplikací vyžaduje řešení problémů, které lze typicky popsat soustavou trigonometrických rovnic. Nejobecněji jde o navigační úlohy nejrůznějších druhů. S pomocí dedikovaného procesoru lze tyto řešit mnohem rychleji než s pomocí softwarové emulace neexistujících instrukcí. Implementaci matematického koprocesoru je podrobně veňována kapitola (3).

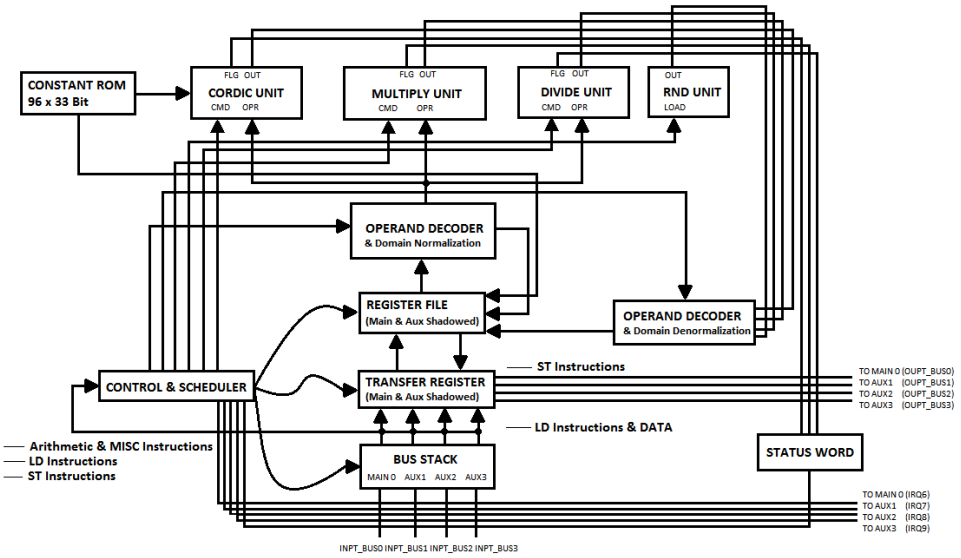
Poznámka. Součástí designu je i integrovaný VGA a PS/2 ovladač, kterému je věnována kapitola (4). Vzhledem k narůstající komplexnosti počítače byly vybrané periférie implementovány z důvodu lepší prezentace výstupů uživateli.

Kapitola 3

FXP koprocesor

V této kapitole bude rozebrán návrh matematického koprocesoru, dále FXP, v notaci pevné čárky. Obecně je FXP univerzální CORDIC procesor. Systém používá originální, nemodifikovaný mechanismus, popsaný v sekci (3.6). Definice instrukcí pak lze nalézt v závěru oddílu.

Níže je uvedeno principiální schéma celého koprocesoru.



3.1 FXP registry

Koprocesor obsahuje celkem 8 aritmetických registrů ARi, které slouží zejména jako zdrojové a cílové operandy početních operací. *Transfer Register* slouží pro přesun dat ze sběrnicového zásobníku do jednotlivých aritmetických registrů. Kontrolní a stavové registry CR a SR slouží ke konfiguraci přerušení a rovněž obsahují stavové vlajky koprocesoru. Celý souborový registr je definován v tabulce níže.

Mnemonics	Name	Width	Mnemonics	Name	Width
ARITHMETIC REGISTERS			**DATA TRANSFER AND CONTROL**		
AR0	Register 0	36	TR	Transfer Register	36
AR1	Register 1	36	CR	Control Register	12
AR2	Register 2	36	SR	Status Register	12
AR3	Register 3	36			
AR4	Register 4	36			
AR5	Register 5	36			
AR6	Register 6	36			
AR7	Register 7	36			

Tabulka 3.1: Přehled FXP registrů

Poznámka. Transfer a aritmetické registry jsou *stínovány* (shadowed). Každá z výkonných jednotek počítače má tedy svůj vlastní, nezavislý souborový registr MAIN, AUX1, AUX2 a AUX3.

- Sběrnicový zásobník vyobrazený na schématu výše je ve skutečnosti 16-ti elementová fronta. Extra kódy XTR a data jsou zařazeny plánovačem vždy na konec fronty.
- Vrch fronty obsahuje aktuální instrukční či datové slovo k vyčtení. Instrukce jsou vyčteny dle modifikovaného round-robin algoritmu a z hlediska výkonných jednotek vykonány nezávisle na pořadí (*out of order*). Výsledky jsou uloženy ve *stínovaných* aritmetických registrech příslušné výkonné jednotky. Algoritmus plánovače je rozebrán v oddíle (3.5.1).

CR : CONTROL REGISTER **Write: SCSW** **Read: LCSW**

MSB											**LSB**
RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	IEM	PM	OM	ZM	BM	DM

RSVD	CR.05-11	NI	**Reserved.** Čteno a zapisováno jako nula.
IEM	CR.05	R/W	**Interrupt Enable Mask.** Hlavní vypínač. Nastavení IEM = 0 potlačí veškeré individuální zdroje přerušení a kompletně vypne celý subsystém koprocesoru.
PM	CR.04	R/W	**Precision Mask.** Nastaví vlajku přerušení PE, pokud je detekována ztráta přesnosti.
OM	CR.03	R/W	**Overflow Mask.** Nastaví vlajku přerušení OE, pokud je detekováno přetečení.
ZM	CR.02	R/W	**Zero Divide Mask.** Nastaví vlajku přerušení ZE, pokud je detekován pokus o dělení nulou.
BM	CR.01	R/W	**Bus Overrun Mask.** Nastaví vlajku přerušení BE, pokud je detekován pokus o zápis dat do již plného sběrnicového zásobniku.
DM	CR.00	R/W	**Done Mask.** Nastaví přerušení kdykoliv je dokončena trigonometrická, hyperbolická nebo aritmetická[1] operace.

Tabulka 3.2: FXP Control Register

SR : STATUS REGISTER			**Write: SCSW**							**Read: LCSW**
MSB										**LSB**
RSVD	IR0	IR1	IR2	IR3	PE	OE	ZE	BE	NF	ZF	BSY

RSVD	SR.11	NI	**Reserved.** Čteno a zapisováno jako nula.
IR0	SR.10	R/W	**Interrupt Request.** Vlajka indikuje přítomnost neobslouženého zdroje přerušení MAIN jednotky.
IR1	SR.09	R/W	**Interrupt Request.** Vlajka indikuje přítomnost neobslouženého zdroje přerušení AUX1 jednotky.
IR2	SR.08	R/W	**Interrupt Request.** Vlajka indikuje přítomnost neobslouženého zdroje přerušení AUX2 jednotky.
IR2	SR.07	R/W	**Interrupt Request.** Vlajka indikuje přítomnost neobslouženého zdroje přerušení AUX3 jednotky.
PE	SR.06	R/W	**Precision Exception.** Vlajka zdroje přerušení detekce ztráty přenosti výsledku.
OE	SR.05	R/W	**Overflow Exception.** Vlajka zdroje přerušení detekce přetečení výsledku.
ZE	SR.04	R/W	**Zero Divide Exception.** Vlajka zdroje přerušení detekce dělení nulou.
BE	SR.03	R/W	**Bus Exception.** Vlajka zdroje přerušní *Bus overrun.*
NF	SR.02	R/W	**Negative Flag.** Vlajka zdroje přerušení je nastavena, pokud poslední výsledek výpočtu je záporný.
ZF	SR.01	R/W	**Zero Flag.** Vlajka zdroje přerušení je nastavena, pokud poslední výsledek výpočtu je nula.
BSY	SR.00	R/W	**Busy Flag.** Stavová vlajka BSY je nastavena, pokud je CORDIC jednotka zaneprázdněna.

Tabulka 3.3: FXP Status Register

SR a CR registry nejsou paměťově mapovány. K jejich čtení a zápisu je nutno využít dedikovaných instrukcí SCSW a LCSW, které jsou v textu definovány dále.

3.2 Číselný formát

Implementovaný matematický koprocesor interpretuje veškeré numerické hodnoty jako posloupnost bitu konečné délky v reprezentaci pevné desetiné

čárky – odtud označení FXP neboli FiXed Point[2]. Zatímco hlavní a auxiliarní jednotky operují na číslech o *libovolné přesnosti* (arbitrary precision) řazených do 12–bitových slov, FXP koprocesor provádí výpočty pouze na 36-bitových **Single Precision Fixed Point** slovech.

MSB					**LSB**
35	...	33	32	...	0
	CONF			**NUM**	

- **Radix Configuration (CONF)**. Označuje jednu z možných konfigurací *destinné* čárky. Měřítko tak lze měnit v závislosti na dané matematické operaci, kdy je potřeba vhodně přizpůsobit počet bitů integrální a zlomkové části čísla. Měřítko je rovněž automaticky změněno při pokusu o provedení instrukce na operandech s rozdílnými konfiguracemi. Nejnižší LSB bity jsou koprocesorem vždy vynulovány (*truncated*).

- **Number (NUM)**. Numerická hodnota v notaci dvojkového doplňku.

Poznámka. Pole CONF konfigurace desetinné čárky rozhodně nemá za cíl *napodobovat* vlastnosti *plovoucí* číselné reprezentace, nýbrž pouze umožnit koexistenci přesnějších CORDIC výpočtů spolu s izolovanou podporou větších čísel, například při součtu či násobení.

Níže jsou uvedeny dostupné konfigurace spolu s odpovídajícím rozsahem.

BASE 10 SCALE RATIOS		
± 4.294967295	10^{-9}	Trigonometrické a hyperbolické operace
± 4294.967295	10^{-6}	Ostatní operace

Namísto více rozšířenému binárnímu formátu používá FXP poněkud komplikovanější desetinný formát. Pokud daný řetězec bitů představující danou numerickou hodnotu budeme chápat jako binární, pak změnu měřítka, neboli posun *binární* tečky, provedeme prostým bitovým posunem.

Obdobně pak postupujeme i v případě posloupnosti bitů představující numerickou hodnotu o základu deset; nyní však musí být každý posun *desetiný*, tj. posun o desetinné místo doleva či doprava. Koprocesor má k těmto účelům dedikovány instrukce SCALE, DSL a DSR – zhlediska uživatelského softwaru je tudíž výkonová penalizace oproti binární reprezentaci minimální.

Podrobněji se otázkou implementace desetinného posunu zabývá podkapitola (3.7). Definice instrukcí posunu lze najít v (3.9.4).

[2]Název byl zvolen z důvodu jednoduchého odlišení od typicky používaného pojmu FPU – Floating Point Unit, který by naznačoval reprezentaci v *plovoucí* desetinné čárce.

3.3 Rozhraní FXP

Z důvodu zachování zpětné kompatibility s původní instrukční sadou a architekturou datové cesty počítače jsou koprocesorové instrukce implementovány jako tzv. *extra kódy* XTR. Nejvyšší, původně rezervovaný *don't care* bit instrukčního slova PIO (Process Input and Output) nyní slouží k odlišení všech XTR instrukcí, jak je patrné z kodového schématu níže.

PIO	X	X	X	X	X	X	X	X	X	Z	Y	0	OPR	1	1	0

ORIGINAL PIO (PROCESS INPUT/OUTPUT) ENCODING

PIO	MOD	X	X	X	X	X	X	X	X	Z	Y	0	OPR	1	1	0

CURRENT PIO (PROCESS INPUT/OUTPUT) ENCODING

Tabulka 3.4: Formát extra kódu XTR

- PIO s MOD = 0 bude směrována příslušné výkonné jednotce.
- PIO s MOD = 1 bude směrována matematickému koprocesoru FXP.

MOD	**OPR**	**TYPE**[3]	**PROCESSOR**[4]
0	0	PIO IO	MAIN
0	1	PIO A	MAIN
1	0	XTR	FXP
1	1	XTR	FXP

FXP *Extra kódy* XTR ovládají FXP, všechny ostatní instrukce jsou zpracovány přímo výkonnou jednotkou. K přenesení výpočetních dat do matematického koprocesoru slouží následující procedura:

1. Zdrojové operandy musí být v první řadě přeneseny do datové paměti.
2. Transfer register TR je naplňen buď *okamžitými* (immediate) nebo *nepřímými* daty (indirect). LD automaticky zkopíruje 36 bitů (tři 12-bitová slova) z dané zdrojové adresy do Transfer registru.
3. Data z TR jsou přemístěny pomocí TRA instrukce do zvoleného cílového aritmetického registru AR0 . . . AR7.
4. Počítač je nyní připraven vykonat danou matematickou operaci na aritmetických registrech AR0 . . . AR7.

K přenesení dat z FXP do počítače je nutno vykonat následující proceduru:

1. Po dokončení CORDIC nebo MPY operace je vygenerováno přerušení pro danou jednotku. Tento krok je automatický. Vektorování přerušení

[3] Všechny FXP instrukce jsou *Extra kódy* neboli XTR.
[4] MAIN zde označuje hlavní procesor se všemi výkonnými jednotkami.

lze potlačit vymaskováním příslušných zdrojových bitů v CR registru nebo stínovaném IR registru.

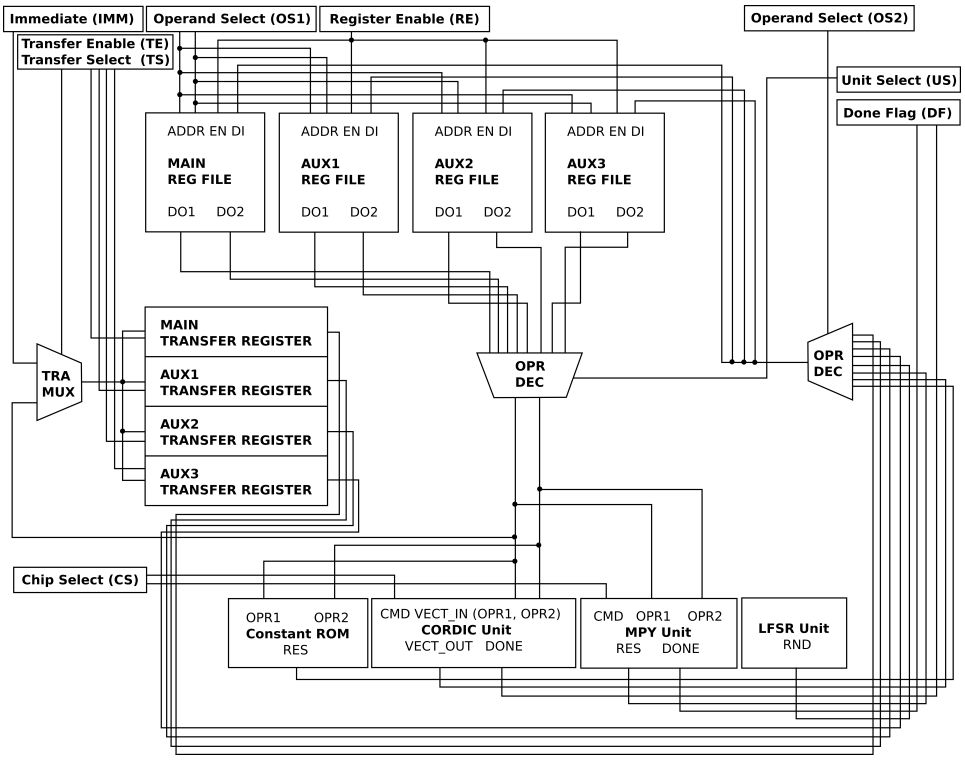
2. Obsah zvoleného aritmetického registru AR*i* je pomocí TRA instrukce přesunut zpět do Transfer registru TR.

3. Obsah Transfer registru TR může být nyní zkopírován do datové paměti na adresu specifikovanou *okamžitým* nebo *nepřímým* operandem. ST automaticky zapíše 36 bitů (tři 12-bitová slova) do datové paměti.

Poznámka. Všechny FXP instrukce s výjimkou datových přesunů LD a ST jsou *neblokující* (non-blocking). Daná výkonná jednotka počítače jednoduše zařadí FXP instrukci do sběrnicové fronty a pokračuje v běhu programu. Podrobnosti o instrukcích přesunu lze nalézt v (3.9.4).

3.4 Datová cesta

Datová cesta koprocesoru sestává z *stínovaného* registrového souboru, dekodérů zdrojových a cílových operandů a příslušných funkčních jednotek. Schematicky je toto zapojení znázorněno níže.



Obrázek 3.1: Datová cesta koprocesoru

<table>
<tr><td>CS.0</td><td>Chip Select. Prostřednictvím tohoto signálu je aktivována či deaktivována CORDIC nebo MPY jednotka. Po aktivaci komponenty jsou ihned automaticky registrovány vstupy vybrané jednotky.</td></tr>
<tr><td>OS0.0–2</td><td>Operand Select. Tento signál ovládá adresový vstup příslušného souborového registru. Signál přepíná první zdrojové operandy OPR1 aritmetických registrů, které jsou pak pomocí operandového dekodéru připojeny na vstupy jednotek CORDIC, MPY a ROM paměti s předpočtenými konstantami. Možné hodnoty, kterých může OS0.0–2 nabývat, jsou uvedeny v tabulce níže.
<table>
<tr><th>OS0/1.0</th><th>OS0/1.1</th><th>OS0/1.2</th><th>Aritmetický registr</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>AR0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>AR1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>AR2</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>AR3</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>AR4</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>AR5</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>AR6</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>AR7</td></tr>
</table></td></tr>
<tr><td>OS1.0–2</td><td>Operand Select. Tento signál ovládá adresový vstup příslušného souborového registru. Signál přepíná druhé zdrojové operandy OPR2 aritmetických registrů a zároveň vybírá cílový aritmetický registr ve fázi zpětného zápisu writeback. Hodnoty kterých může OS1.0–2 nabývat v případě výběru druhého operandu je totožný s těmi uvedenými v tabulce pro OS0.0–2.</td></tr>
<tr><td>OS2.0–4</td><td>Operand Select. Tento signál ovládá cílový operandový dekodér, jehož výstupy jsou přivedeny na vstupy souborového registru. Hodnoty, kterých může signál nabývat jsou uvedeny v tabulce níže.
<table>
<tr><th>OS2</th><th>Zdroj</th><th>OS2</th><th>Zdroj</th></tr>
<tr><td>0</td><td>CORDIC x Output</td><td>7</td><td>RND Output</td></tr>
<tr><td>1</td><td>CORDIC y Output</td><td>8</td><td>Immediate</td></tr>
<tr><td>2</td><td>CORDIC z Output</td><td>9</td><td>MAIN Transfer</td></tr>
<tr><td>3</td><td>MPY Lo Output</td><td>10</td><td>AUX1 Transfer</td></tr>
<tr><td>4</td><td>MPY Hi output</td><td>11</td><td>AUX2 Transfer</td></tr>
<tr><td>5</td><td>Reserved</td><td>12</td><td>AUX3 Transfer</td></tr>
<tr><td>6</td><td>ADD Output</td><td>13</td><td>OPR1 Output</td></tr>
</table></td></tr>
</table>

US.0	**Unit Select.** Ovládá demultiplexor operandového dekodéru a připojuje výstupy OPR1 a OPR2 na příslušné vstupy jednotek CORDIC, MPY a ROM paměti s předpočtenými konstantami.
RE.0–3	**Register Enable.** Ovládá zápis do příslušného aritmetického registru ARi. Pokud RE.$i = 1$ obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna.
TE.0	**Transfer Enable.** Ovládá zápis do příslušného transfer registru TRA. Pokud TRA = 1 obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna.
TS.0	**Transfer Select.** Přepíná mezi zdroji transfer multiplexoru, a to mezi registrem okamžité hodnoty IMM a operandem OPR1.
IMM.35–0	**Immediate.** Obsahuje výstup z registru okamžité hodnoty IMM, který je přiveden na vstup multiplexoru pro *stínované* Transfer registry jednotlivých výkonných jednotek. IMM *není* kontrolní signál.

Tabulka 3.6: Přehled kontrolních signálů

3.5 Kontrolní logika

Kontrolní logika koprocesoru generuje všechny potřebené signály nutné k obsluze datové cesty a rovněž zahrnuje plánovač sběrnicové fronty. Jelikož byly signály datové cesty definovány v sekci (3.4), omezíme se zde pouze na definici instrukčního dekodéru. Ten můžeme pomocí *psuedokódu* definovat takto:

```
switch(OPCODE)
{
  case 0x0:
  switch(FNC)
  {
    case 0: state = SINCOSH;
    case 1: state = ATAN;
    case 2: state = ATANH;
    case 3: state = SINHCOSH;
    case 4: state = TAN;
    case 5: state = TANH;
    case 6: state = SQRT;
```

```
    case 7: state = LN;
  }
  case 0x1: state = ADD;
  case 0x2:
    switch(MOD)
    {
      case 0:  state = COM;
      case 1:  state = LDS;
      case 2:  state = RDS
      default: state = Computer Alarm
    }
  case 0x3: state = RND;
  case 0x4: state = DIV;
  case 0x5: state = MPY;
  case 0x6: state = CMP;
  case 0x7: state = LD;
  case 0x8: state = ST;
  case 0x9: state = LDC;
  case 0xA: state = TRA;
  case 0xB:
    switch(MOD)
    {
      case 0:  state = XWAIT;
      case 1:  state = ABS;
      case 2:  state = SCALE
      default: state = Computer Alarm
    }
  case 0xC: state = LCSW;
  case 0xD: state = SCSW;
  case 0xE: state = MOV;
  default:  state = Computer Alarm
}
```

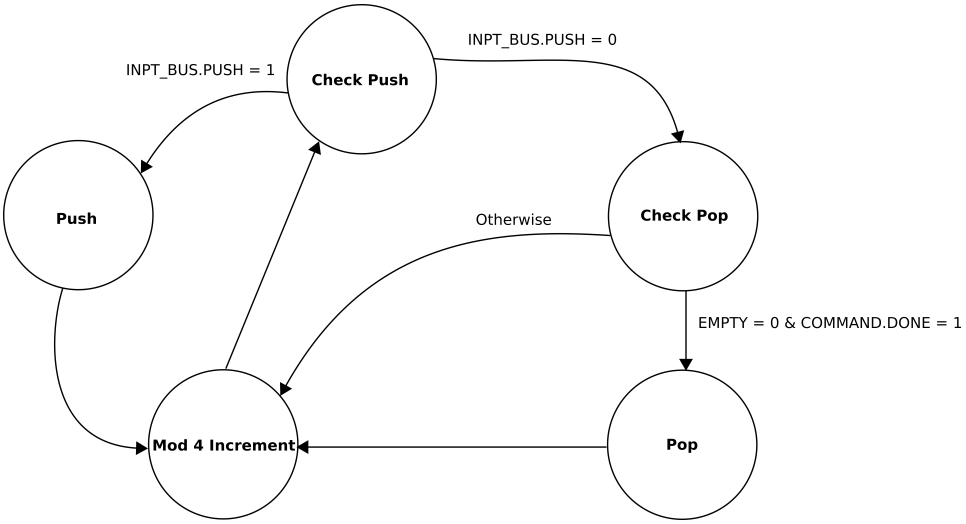
Tabulka 3.7: Instrukční dekodér koprocesoru

Poznámka. Definice bitových polí FNC a MOD lze v textu nalézt dále a to v oddílu (3.9).

3.5.1 Plánovač sběrnicové fronty

Data mezi počítačem a koprocesorem jsou směňována prostřednictvím *full-duplexní* sběrnice. Data je možné konkuretně zapisovat zpět a do koproce-

soru, nicméně kontrolní jednotka koprocesoru může vyčítat data pouze v jediném, přesně daném okamžiku. Tento mechanismus implementuje právě *plánovač sběrnicové fronty*, jehož principiální schéma je uvedeno níže.



Obrázek 3.2: Plánovač sběrnicové fronty

Plánovač je řízen následujícimi signály:

Signál	Bus type	Poznámka
PUSH	Input Bus	Indikuje zda počítač zapisuje data. Prioritní signál.
DONE	Input Bus	Tento signál je v nízké úrovni, pokud koprocesor nevykonává žádnou instrukci. Instrukce vyčtená v minulém čase byla právě dokončena.
EMPTY	Interní[5]	Tento signál je ve vysoké úrovni, pokud je sběrnicová fronta prázdná. Koprocesor nemá žádná data k zpracování.
OKAY	Output Bus	Tímto signálem potvrzuje koprocesor úspešný příjem dat ze sběrnice. Tento signál je automaticky resetován v *Check Push* stavu.

Instrukce může být z datové fronty vyčtena *pouze* pokud daná výkonná jednotka počítače zrovna nezapisuje žádná data, tj. pokud je signál **PUSH** $= 0$ a pokud koprocesor sám není již zaneprázdněn vykonáváním jiné instrukce **DONE** $= 1$.

Poznámka. *Check Push* označuje výchozí stav.

[5]Interní signál fronty plánovače. Signál není součástí sběrnice.

3.6 CORDIC

CORDIC neboli, **CO**ordinate **R**otational **DI**gital **C**omputer, je *shift-and-add* algoritmus původně vyvinutý pro výpočet trigonometrických a hyperbolických funkcí. Hluboce *pipelinenovaný* CORDIC je nenáročný na plochu chipu a samotný algoritmus užívá k výpočtu pouze operací součtu, bitového posunu a vyčtení předem tabulovaných výpočtů (*table lookup*).

Jedna z původních výzkumných prací[4] zmiňuje následující rovnice, které vycházejí z úhlových součtových vzorců a dalších identit:

$$K_i R \sin(\theta \pm \phi) = R \sin(\theta) \pm 2^{-i} R \cos(\theta) \tag{3.1}$$

$$K_i R \cos(\theta \pm \phi) = R \cos(\theta) \mp 2^{-i} R \sin(\theta) \tag{3.2}$$

kde $K_n = \sqrt{1 + 2^{-2i}}, i \in \mathbb{N}$. Výše uvedené rovnice mohou být užity k rotování vektoru R o kladný či záporný inkrement $\tan\left(2^{-i}\right)$. Iterativním výpočtem rovnic (7.1) s roustoucí mocninou $i \to \infty$ je tak možné dosáhnout rotace vektoru R o libovolný úhel.

K řešení úlohy používá CORDIC fundamentální princip rotace vektoru o daný inkrement úhlu. Začněme tedy neprve s definicí rotační matice $R(\theta)$ ve dvou rozměrech:

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{3.3}$$

Vektor $\boldsymbol{v} = (x, y)$ pak můžeme rotovat kolem počátku souřadného systému o úhel θ užitím maticového násobení.

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \boldsymbol{v}_{i+1} = R\boldsymbol{v}_i \tag{3.4}$$

kde $\boldsymbol{v}_{i+1} = (x_{i+1}, y_{i+1})$ je nový vektor vzniklý rotací $\alpha_{i+1} = \alpha_i + \theta$ a α_i značí počáteční úhel vektoru $\boldsymbol{v}_i$ v i-té iteraci. Rotovaný úhel θ tak vlastně konstruujeme pomocí součtu dílčích úhlových inkrementů:

$$\theta = \alpha_1 + \alpha_2 + \ldots + \alpha_n \tag{3.5}$$

kde $n \in \mathbb{N}$ je konečný počet iterací. Rozepsáním (3.4) dostaneme vztahy pro jednotlivé komponenty vektoru $\boldsymbol{v}_{i+1}$:

$$x_{i+1} = x_i \cos\theta - y \sin\theta \tag{3.6}$$

$$y_{i+1} = y_i \sin\theta + y \cos\theta \tag{3.7}$$

Rotace o libovolný úhel θ je však netriviální, neboť neznáme příslušné prvky rotační matice $R(\theta)$. Nejprve zavedeme následující trigonometrické identity:

$$\cos\theta = \frac{1}{\sqrt{1 + \tan^2\theta}} \tag{3.8}$$

$$\sin\theta = \frac{\tan\theta}{\sqrt{1 + \tan\theta}} \tag{3.9}$$

S užitím výše uvedených identit lze rotační matici $R(\theta)$ přepsat do tvaru

$$R(\theta) = \frac{1}{\sqrt{1+\tan^2\theta}} \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix} \tag{3.10}$$

Pokud nyní omezíme obor hodnot úhlu θ pouze na $\pm 2^{-i}$ můžeme počáteční vztah rotace (3.4) přepsat jako

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \frac{1}{\sqrt{1+2^{-2i}}} \begin{pmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{3.11}$$

kde d_i nabývá v daném iteračním kroku i hodnot ± 1 a to v závislosti na úhlu θ:

$$d_i = \begin{cases} +1 & \theta \geq 0 \\ -1 & \text{jinde} \end{cases} \tag{3.12}$$

Hodnota d_i tedy určuje směr rotace vektoru. Tento krok je naprosto kritický pro výpočetní mechaniku CORDICu, neboť umožňuje zcela eliminovat násobení funkcí $\tan(\theta_i)$ a nahradit ji mocniným dělením o základu 2, které lze digitálně snadno implementovat pomocí bitových posunů.

$$\arctan 2^{-i} \approx 2^{-i} \quad i \to \infty \tag{3.13}$$

Vztah (3.13) je důsledkem následující identity:

$$\lim_{i \to 0} \frac{\arctan(x)}{x} = 1 \tag{3.14}$$

Funkce $\arctan(x)$ je monotónní a rostoucí na intervalu $[0, \pi/2)$ a v $x = 0$ nabývá její derivace hodnoty 1.

$$\frac{\mathrm{d}}{\mathrm{d}\,x} \arctan(x)\bigg|_{x=0} = \lim_{\Delta \to 0} \frac{\arctan(x+\Delta)}{x+\Delta-x}\bigg|_{x=0} = 1 \tag{3.15}$$

Proto (3.14) a (3.16) jsou ekvivalentní

$$\lim_{i \to +\infty} \frac{\arctan\left(2^{-i}\right)}{2^{-i}} = 1 \tag{3.16}$$

Rozepsáním maticového součinu (3.11) do složek nyní dostaneme

$$x_{i+1} = K_i(x_i - d_i 2^{-i} y_{i-1}) \tag{3.17}$$

$$y_{i+1} = K_i(y_i + d_i 2^{-i} x_{i-1}) \tag{3.18}$$

kde $K_i = \frac{1}{\sqrt{1+2^{-2i}}}$ a označuje *zisk* rotace provedené v i-té iteraci. Můžeme si povšimnout, že výše uvedeným postupem se nám podařilo rotovat vektor $\boldsymbol{v}$

o úhel θ, nicméně za cenu navýšení jeho magnitudy v každé iteraci i o faktor K_i. Konečný vektor tedy po n iteracích nabyde velikosti o faktor

$$K = \prod_{i=0}^{n} \sqrt{1 + \tan^2 \alpha_i} = \lim_{n \to \infty} \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}} \tag{3.19}$$

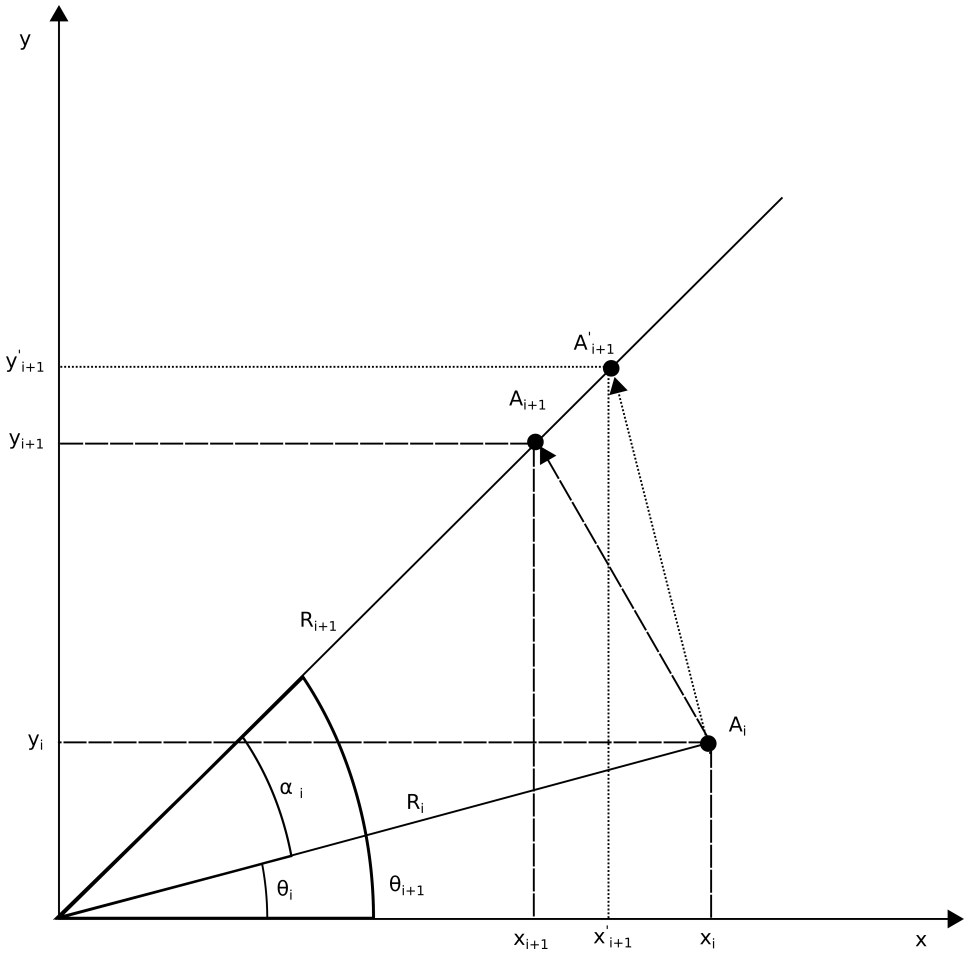
Pro konstatní počet iterací je však K známé a vynásobením rotovaného vektoru převrácenou hodnotu $1/K$ lze tento jev zcela eliminovat.

CORDIC je požíván ve dvou konfiguracích:

- **Rotace.** Výchozí vektor je $\boldsymbol{v} = (1, 0)$, který bude rotován kolem počátku souřadnicového systému o úhel θ. Velikost cílového vektoru lze v tomto režimu kompenzovat hned v počátku a to dosazením konstanty $1/K$ za x_0, tzn. $\boldsymbol{v} = (1/K, 0)$. Rotace zde má význam $(\cos\theta, \sin\theta)$.

- **Vektorování.** Vychozí vektor $\boldsymbol{v} = (x, y)$ bude rotován kolem počátku souřadného systému o úhel θ. Vektorovací režim je modifikací rotačního režimu. Vektor $\boldsymbol{v}$ je rotován, dokud není komponenta $y = 0$. Rotace zde ma význam $\theta = \arctan(y/x)$. Kalkulace zároveň určí i magnitudu vektoru $||\boldsymbol{v}|| = \sqrt{x^2 + y^2}$. Režim vektorováni tudíž umožňuje okamžitý převod z kartézkých do polárních souřadnic.

Na obrázku níže je zobrazen proces jedné elementární rotace CORDICu. Počáteční úhel θ_i je v i–té iteraci rotován o úhel α_i. Konečný úhel je θ_{i+1}. Všimněme si však, že magnituda původního vektoru byla navýšena z R_i na R_{i+1}. Výchozí bod A_i byl tudíž posunut do bodu ozačeném jako A'_{i+1}, zatímco skutečná velikost vektoru by správně měla zůstat nezměněna, tj. cílový bod je A_{i+1}. Říkáme, že rotace má *zisk* K popsaný výše. Tento zisk je po dokončení iterativní rotace nutno kompenzovat.



Obrázek 3.3: Princip CORDIC mechanismu

Výše uvedené rovnice lze dále zobecnit pro lineární, kruhové a hyperbolické souřadnice. FXP jednotka implementuje následující universální CORDIC rovnice:

$$x_{i+1} = x_i - \mu d_i y_i 2^{-i} \tag{3.20}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{3.21}$$

$$z_{i+1} = z_i - d_i \alpha_i \tag{3.22}$$

MODE	**Rotation**	**Vectoring**
	$d_i = \mathrm{sgn}(z_i), \quad z \to 0$	$d_i = -\mathrm{sgn}(x_i y_i), \quad y \to 0$
Circular	$x_{i+1} = K(x_i \cos z_i - y_i \sin z_i)$	$x_{i+1} = K\sqrt{x_i^2 + y_i^2}$
$\mu = 1$	$y_{i+1} = K(y_i \cos z_i - x_i \sin z_i)$	$y_{i+1} = 0$
$\alpha_i = \tan^{-1} 2^{-i}$	$z_{i+1} = 0$	$z_{i+1} = z_i + \tan^{-1}(y_i/x_i)$
Linear	$x_{i+1} = x_i$	$x_{i+1} = x_i$
$\mu = 0$	$y_{i+1} = y_i + x_i z_i$	$y_{i+1} = 0$
$\alpha_i = 2^{-i}$	$z_{i+1} = 0$	$z_{i+1} = z_i + y_i/x_i$
Hyperbolic	$x_{i+1} = K'(x_i \cosh z_i - y_i \sinh z_i)$	$x_{i+1} = K\sqrt{x_i^2 + y_i^2}$
$\mu = -1$	$y_{i+1} = K'(y_i \cosh z_i - x_i \sinh z_i)$	$y_{i+1} = 0$
$\alpha_i = \tanh^{-1} 2^{-i}$	$z_{i+1} = 0$	$z_{i+1} = z_i + \tanh^{-1}(y_i/x_i)$

Tabulka 3.9: Univerzálnín rovnice CORDIC

Poznámka. Znaménková funkce $\mathrm{sgn}(x)$ užitá v rovnicích (3.20), (3.21) a (3.22) je poněkud odlišná od standardní funkce $\mathrm{sgn}(x)$ užívané v matematice. Ve výše uvedených rovnicích ji tedy definujeme takto

$$\mathrm{sgn(x)} = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{3.23}$$

Znaménková funkce (3.23) musí nabývat hodnoty 1 pro $x = 0$, neboť značí principiální *rozhodovací* funkci CORDICu – pokud je daný úhel po rotaci větší než úhel hledaný, je nutno kompenzovat tuto situaci opačnou rotací v následující iteraci a naopak.

3.6.1 Předpočítané konstanty

Níže jsou uvedeny veškeré předpočtené úhly α_i užité v rovnicích (3.22). Úhly byly nejprve vypočteny v plovoucí desetiné čárce nativní 80-bitové *Extra přesnosti* x87 kompatibilního FPU procesoru a následně převedeny do notace pevné desetinné čárky o přesnosti 10^{-9}. Počátkem iterace $i = 30$ je již úhel příliš malý, než aby šel reprezentovat formátem (3.2). Teoretická přesnost CORDIC jednotky je tudíž omezena na 10 platných číslic.

i	Angle [Radians]	i	Angle [Radians]
ARCTAN ANGLES LOOKUP TABLE			
0	785398163	16	15258
1	463647609	17	7629
2	244978663	18	3814
3	124354994	19	1907
4	62418809	20	953
5	31239833	21	476
6	15623728	22	238
7	7812341	23	119
8	3906230	24	59
9	1953122	25	29
10	976562	26	14
11	488281	27	7
12	244140	28	3
13	122070	29	1
14	61035	30	0
15	30517	31	0
ARCTANH ANGLES LOOKUP TABLE			
0	NaN[6]	16	15258
1	549306144	17	7629
2	255412811	18	3814
3	125657214	19	1907
4	62581571	20	953
5	31260178	21	476
6	15626271	22	238
7	7812658	23	119
8	3906269	24	59
9	1953127	25	29
10	976562	26	14
11	488281	27	7
12	244140	28	3
13	122070	29	1
14	61035	30	0
15	30517	31	0

Tabulka 3.11: Předpočítané tabulky úhlů atan(x) a atanh(x)

3.6.2 Konvergence

Mechanimus CORDICu vyprodukuje platné výstupy pouze pro určitý vstupní interval. V tomto oddíle zmíníme podmínky konvergence pro kruhové

[6] *Not a Number*, záznam $\tanh^{-1}(2^0)$ je přeskočen.

a hyperbolické souřadnice.

Kruhové souřadnice

Obecná podmínka konvergence rotace v i–té iteraci musí splňovat následující relaci[8]:

$$|z[i]| \leq \sum_{j=i}^{+\infty} \arctan\left(2^{-j}\right) \tag{3.24}$$

Symbol $z[i]$ zde tedy reprezentuje výsledný úhel, který je výsledkem i elementárních rotací. Suma na pravé straně výrazu označuje součet zbývajích elementárních úhlových rotací potřebný k dokončení celkové rotace o úhel θ. Ve skutečnosti je potřeba nekonečné množství iterací pro dosažení plné rotace úhlu. Teoretický maximální vstupní úhel $\theta_{\max}$, o který lze rotovat je tudíž roven,

$$\theta_{\max} = \sum_{j=0}^{+\infty} \arctan\left(2^{-j}\right) \approx 1.7429\,[\text{rad}] \tag{3.25}$$

což je součet všech elementárních úhlů. V reálné implementace je však možné realizovat pouze konečný počet iterací n. V takovém případě lze pro úhel $\theta_{\max}$ psát:

$$\theta_{\max} = \sum_{i=0}^{n-1} \arctan\left(2^{-i}\right) + 2^{-n+1} \tag{3.26}$$

kde 2^{-n+1} je tzv. *residuální úhel*, tedy rozdíl mezi výsledným úhlem po n iterací a úhlem cílovým. Residuální úhel vzniká z důvodu nemožnosti splnění podmínky (3.24). Počítač má vždy pouze omezenou paměť, která nemůže obsáhnout všechny elemenátrní úhlové rotace.

Hyperbolické souřadnice

Obdobný princip lze aplikovat i na hyperbolické souřadnice. Funkce $\arctan(x)$ je pouze nahrazena její hyperbolickou obdobou a ze stejného důvodu je posunut index sumace o jedničku.

$$|z_0| \leq \operatorname{arctanh}(2^{-n}) + \sum_{i=1}^{n} \operatorname{arctanh}(2^{-i}) \tag{3.27}$$

Teoretický maximální vstupní úhel $\theta_{\max}$ pro nekonečný počet iterací $n \to +\infty$ tedy je

$$\theta_{\max} \approx 1.182 \tag{3.28}$$

3.7 Součin

3.7.1 MBE

Podívejme se nejprve na původní Boothův algoritmus. Schéma definuje celkem tři registry a to násobitel M (multiplicand), kvocient Q (quotient) a součin P (product). Registr součinu P má charakter akumulátoru (*střadače*), tzn. že akumuluje výsledky předchozích iterací. Označme nyní respektivně symboly i a j počet bitů nutný k reprezentaci skutečného násobitele m a kvocientu q ve dvojkovém doplňku. Boothův algoritmus pak spočte hodnotu $m \cdot q$ a obecně sestává ze dvou klíčových kroků:

1. Na základě analýzy posledních dvou LSB bitů je přičten k akumulátoru registr M nebo Q.

2. Na akumulátor součinu P je aplikován pravý bitový aritmetický posun.

Registry M, Q a P šířky $i + j + 1$ jsou na začátku procedury nastaveny následujícím způsobem:

- Nejvyšší bity MSB registru M jsou naplněny i bity násobitele m, zbývající bity jsou vynulovány.

- Nejvyšší bity MSB registru Q jsou naplněny i bity násobitele m s opačným znaménkem, zbývající bity jsou vynulovány.

- Nejvyšších i MSB bitů registru P je vynulováno. Zbývající LSB bity až na poslední jsou nastaveny na hodnotu kvocientu q. Poslední LSB bit je vynulován. Tento bit nazýváme *implicitním*.

Poznámka. Případný *overflow* plynoucí ze součtů provedených v prvním kroku je vždy ignorován.

První krok algoritmu zpočívá v analýze posledních dvou LSB bitů dle následujícího schématu:

i	$i-1$	Operation	
BIT	**POSITIONS**		
0	0	$\pm 0 \cdot M$	Add 0 (do nothing)
0	1	$+1 \cdot M$	Add M
1	0	$+1 \cdot M$	Add Q
1	1	$+2 \cdot M$	Add 0 (do nothing)

Tabulka 3.12: Původní Boothovo schéma

Kroky 1 a 2 algoritmu jsou opakovány celkem i-krát. Po dokončení i iterací je zahozen implicitní bit. Registr P nyní obsahuje hodnotu součinu $m \cdot q$.

Je patrné, že originální Boothovo schéma vyprodukuje i mezivýsledků. Modifikované Boothovo schéma je navrženo tak, že redukuje počet mezivýsledků

na $i/2+1$. Protože procedura násobení je *pipelinována* rychlostí jedné iterace za hodinový cyklus, zaznamenáme pokles z 32 hodinových cyklů na pouhých 17 v případě 32-bitového násobitele. Toho je docíleno prostřednictvím následujícího MBE mechanismu:

- Namísto dvou bitů jsou analyzovány bity tři. Tyto tři bity MBE označuje jako *triplet* – bitová trojice. Namísto (3.7.1) jsou triplety překládány pomocí schématu (3.12):

$i+1$	i	$i-1$	Operation	
BIT POSITIONS				
0	0	0	$\pm 0 \cdot M$	Add 0 (do nothing)
0	0	1	$+1 \cdot M$	Add M
0	1	0	$+1 \cdot M$	Add M
0	1	1	$+2 \cdot M$	Shift M left and add
1	0	0	$-2 \cdot M$	Shift M left, complement and add
1	0	1	$-1 \cdot M$	Complement M and add
1	1	0	$-1 \cdot M$	Complement M and add
1	1	1	$\pm 0 \cdot M$	Add 0 (do nothing)

Tabulka 3.13: Modifikovné Boothovo schéma

Celý proces je obdobou původního Boothova schématu. Koprocesor implementuje následující proceduru:

1. **Initialize.** Násobitel a kvocient m, q jsou rozšířeny tak, aby měly stejný počet bitů a zároveň aby byl počet bitů sudý. Registr P je vynulován. Registr M je nastaven na hodnotu multiplikandu m. Registr Q je nastaven na hodnotu kvocientu q a následně rozšířen o LSB (*trailing zero extend*), který označíme jako *implicitní* bit Q_{-1}. Iterační proměnná je resetována na hodnotu $n/2$, kde n je bitová šířka registrů M a Q.

2. **Compare and compute.** Bity Q_1, Q_0, Q_{-1} nyní tvoří dříve zmíněný *triplet*, který je porovnán s tabulkou (3.12). Dle hodnoty tripletu je provedena příslušná kalkulace, jejíž výsledek je přičten k akumulátoru P. Opět, jakýkoliv *oveflow* je ignorován.

3. **Shift right.** Sloučený bitový řetězec (P & Q)$(2n+1$ downto $0)$, tedy včetně *implicitního* bitu, je aritmeticky posunut doprava o dvě pozice. Zároveň je dekrementována interační proměnná.

4. **Iterate.** Kroky 2 a 3 jsou opakovány dokud není iterační proměnná rovna 0 (*diminished*).

5. **Output.** *Implicitní* bit je zahozen. Výsledek součinu $m \cdot q$ je pak sloučený bitový řetězec $P\&Q$ v notaci dvojkového doplňku.

Poznámka. Pokud je po inicializaci počet bitů sudý, ale následná kombinace bitového posunu a komplementace by způsobila ztrátu známenkové bitu (a tudíž chybný výsledek), je nutné číslo rozšířit (*zero extend*) o další dva MSB bity. Tímto zůstane zachován znaménkový bit a rovněž požadavek početní sudosti bitového řetězce.

Jednotka násobení koprocesoru má následující vstupy:

Signal	Symbol	Direction	Purpose
UNIT PINOUT			
Clock	CLK	Input	System clock.
Reset	RST	Input	Reset line.
Command	CMD	Input	Command line.
Operand 1	OPR1	Input	First operand M.
Operand 2	OPR2	Input	Second operand Q.
Result	RES	Output	Multiply result P.
Done Flag	DONE	Output	Done indicator.

S užitím operandového dekodéru a Unit Select (US) signálu koprocesor ovládá CMD signál násobící jednotky. Po nastavení CMD = 1 započne jednotka iterativní kalkulaci popsanou výše, po jejímž dokončení je nastavena vlajka DONE = 1. Signál CMD musí být v aktivní úrovni po celou dobu iterace. Výsledek součinu OPR1 · OPR2 pak lze vyčíst z registrovaného signálu RES.

3.7.2 Desetinné posuny

Jelikož koprocesor užívá měřítko o desetinném základu, bylo vhodné implementovat nativní desetinné posuny. Obdobně jako je tomu v notaci binární, kde posun o n bitů doprava představuje násobení hodnotou 2^{-n} (případně dělení 2^n) a doleva hodnotou 2^n, značí desetinný posun násobení 10^{-n} nebo 10^n. Stejného efektu lze samozřejmě dosáhnout prostým užitím MPY instrukce, dané číslo vynásobí 10–ti či reciprociální hodnotou 10^{-1}. Za minimální cenu navýšené plochy chipu se však nabízí jiné řešení a to užití příhodného *shift-and-add* algoritmu.

- Desetinného posunu vlevo lze docílit rozkladem čísla 10 na součet binárních posunů, tedy

$$10 = ((1 \cdot 2^2) + 1) \cdot 2^1 \tag{3.29}$$

  S využitím této identity pak lze vynásobit libovolné celé číslo 10–ti:

$$10 \cdot n = ((n \cdot 2^2) + n) \cdot 2^1 \tag{3.30}$$

  kde $n \in \mathbb{N}^0$.

- Pro desetinný posun vpravo platí obdobný princip, s úsporných důvodů jej zde však popíšeme pomocí *pseudokódu*. Zavedeme pomocnou proměnnou q. Procedura

```
unsigned int q;
q = (n >> 1) + (n >> 2);
q = q + (q >> 4);
q = q + (q >> 8);
q = q + (q >> 16);
q = q >> 3;
if((n - (((q << 2) + q) << 1)) > 9)
{
        q += 1;
}
```

  vydělí dané číslo n 10–ti a výsledek podílu zanechá v q.

Uvedené procedury jsou neporovnatelně rychlejší, než násobení prostřednictvím MPY instrukce. Desetinný bitový posun je dokončen zpravidla ve 2 hodinových cyklech pro levý desetinný posun a 5–ti hodinových cyklech pro pravý desetinný posun.

Podrobné definice instrukce posunu RDS, LDS a SCALE lze dále nalézt v (3.9.4).

3.8 Podíl

Z důvodu úspory plochy chipu je pro účely dělení použita CORDIC jednotka v konfiguraci lineárních souřadnic. Podrobnosti mechanismu lze nalézt v(3.20).

3.9 Instrukční sada

V tomto oddílu je rozebrána instrukční sada koprocesoru. Nejprve definujeme formát samotného instrukčního slova a poté jednotlivé instrukce. Ke konci oddílu jsou pak představeny alternativní názvy (*alias*) některých instrukcí z důvodu mnemonické kolize instrukcí FXP a výkonných jednotek hlavního procesoru.

3.9.1 Instrukční slovo

Instrukční slovo FXP koprocesoru je 10–bitové a je tvořeno bity 14 – 5 hlavního programového slova počítače. FXP instrukční slovo je ukládáno a posléze vyčítáno ze sběrnicové fronty nezávisle na běhu hlavního programu. Všechny instrukce, až na vybrané datové, jsou tudíž *neblokující*

(non-blocking).

Níže je uvedeno kódovací schéma obecného instrukčního slova matematického koprocesoru.

MSB									LSB
FNC/X	FNC/X	FNC/X	OPR	OPR	OPR	OP	OP	OP	OP

FNC	IW.09-07	**Function.** Funkce CORDIC nebo cílový či MOD operand.
OPR	IW.06-04	**Operand.** Zdrojový operand.
OP	IW.03-00	**Opcode.** Operační kód instrukce.

Tabulka 3.15: Formát instrukčního slova

3.9.2 Přehled instrukcí

Instrukční sada samotného matematického koprocesoru čítá 23 XTR instrukcí. Níže jsou však pro pořádek uvedeny spolu se všemi dostupnými XTR kódy, tedy i těmi, které jsou zpracovávány pouze výkonnými jednotkami počítače. V následujícím oddílu pak lze nalézt detailní popis každé individuální XTR instrukce.

Mnemonic	Operand	Description	Cycles	FA[7]
TRIGONOMETRIC OPERATIONS				
SINCOS	REG	Sin and Cos of an angle.	32	Yes
ATAN	REG	Arctan of an angle.	32	Yes
TAN	REG	Tan of an angle.	64	Yes
HYPERBOLIC OPERATIONS				
SINHCOSH	REG	Sinh and Cosh of an angle.	32	Yes
ATANH	REG	Arctanh of an angle.	32	Yes
TANH	REG	Tanh of an angle.	64	Yes
LOGARITHMIC OPERATIONS				
LN	REG	Natural logarithm.	96	Yes
ARITHMETIC OPERATIONS				
ADD	REG	Add registers.	2	Yes
COM	REG	Complement register.	1	Yes
MPY	REG	Multiply registers.	17	Yes
DIV	REG	Divide registers.	32	Yes
COMPARSION DATA TRANSFER OPERATIONS				
CMP	REG	Compare registers.	2	Yes
LD	REG	Load indirect.	–	No
LD	MEM	Load immediate.	–	No
ST	MEM	Store transfer register.	–	No
LDC	REG	Load math constant.	3	No
TRA	REG	Transfer to arithmetic.	2	No
MOV	REG	Move registers.	2	No
CNDX	IMM	Change index.	2	No
XCHG	None	Exchange A and B.	4	No
MISCELLANEOUS OPERATIONS				
RND	REG	Generate random number.	2	No
ABS	REG	Absolute value.	32	Yes
SQRT	REG	Square root.	32	Yes
WAIT	None	Wait for computation completion.	–	No
LCSW	MEM	Load Control and Status Word.	–	Yes
SCSW	MEM	Store Control and Status Word.	–	No
SCALE	REG	Change precision scale.[8]	5	No
LDS	REG	Left Decimal shift.	3	Yes
RDS	REG	Right Decimal shift.	5	Yes
RETI	None	Interrupt return.	5	No
SIW	REG	Store interrupt word.	3	No
LIW	REG	Load interrupt word.	3	No

Tabulka 3.16: Přehled XTR instrukcí

3.9.3 Kódování instrukcí

TRIGONOMETRIC OPERATIONS										
SINCOS	0	0	0	OPR	OPR	OPR	0	0	0	0
ATAN	0	0	1	OPR	OPR	OPR	0	0	0	0
TAN	1	0	0	OPR	OPR	OPR	0	0	0	0
HYPERBOLIC OPERATIONS										
SINHCOSH	0	1	1	OPR	OPR	OPR	0	0	0	0
ATANH	0	1	0	OPR	OPR	OPR	0	0	0	0
TANH	1	0	1	OPR	OPR	OPR	0	0	0	0
LOGARITHMIC OPERATIONS										
LN	1	1	1	OPR	OPR	OPR	0	0	0	0
ARITHMETIC OPERATIONS										
ADD	OPR	OPR	OPR	OPR	OPR	OPR	0	0	0	1
COM	X	0	0	OPR	OPR	OPR	0	0	1	0
MPY	OPR	OPR	OPR	OPR	OPR	OPR	0	1	0	1
DIV	OPR	OPR	OPR	OPR	OPR	OPR	0	1	0	0
COMPARSION AND DATA TRANSFER OPERATIONS										
CMP	OPR	OPR	OPR	OPR	OPR	OPR	0	1	1	0
LD	IMM	IMM	IMM	IMM	IMM	OPR	0	1	1	1
ST	IMM	IMM	IMM	IMM	IMM	OPR	1	0	0	0
LDC	X	IMM	IMM	OPR	OPR	OPR	1	0	0	1
TRA	MOD	X	X	OPR	OPR	OPR	1	0	1	0
MOV	OPR	OPR	OPR	OPR	OPR	OPR	1	1	1	0
CNDX	0	0	0	X	X	OPR	1	1	1	1
XCHG	1	0	0	X	X	X	1	1	1	1
MISCELLANEOUS OPERATIONS										
RND	X	X	X	OPR	OPR	OPR	0	0	1	1
ABS	0	1	OPR	OPR	OPR	OPR	1	0	1	1
SQRT	1	1	0	OPR	OPR	OPR	0	0	0	0
WAIT	0	0	X	X	X	X	1	0	1	1
LCSW	MEM	MEM	MEM	MEM	MEM	MEM	1	1	0	0
SCSW	MEM	MEM	MEM	MEM	MEM	MEM	1	1	0	1
SCALE	1	0	IMM	OPR	OPR	OPR	1	0	1	1
LDS	X	0	1	OPR	OPR	OPR	0	0	1	0
RDS	X	1	0	OPR	OPR	OPR	0	0	1	0
RETI	1	1	1	X	X	X	1	0	1	1
SIW	0	1	0	X	X	X	1	1	1	1
LIW	1	1	0	X	X	X	1	1	1	1

Tabulka 3.17: Přehled kódování jednotlivých XTR instrukcí

3.9.4 Definice instrukcí

This sections defines all the extra code (XTR) instructions previously declared.

SINCOS

Funkce	Sinus a Cosinus
Slova	1
Cyklů	32
Kódování	0 0 0 OPR OPR OPR 0 0 0 0

Poznámka. Spočte funkce sinus a cosinus $(\sin, \cos)$ zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR1 a AR0, respektivně. Zdrojový úhel musí být zadán v radiánech a ležet v intervalu $[-1.74; +1.74]$. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přenosti z důvodu nekonvergence algoritmu a vlajka Precision Exception (PE) je nastavena. V takovém případě zůstávají cílové operandy AR1 a AR2 nezměněny.

ATAN

Funkce	Inverzní tangens
Slova	1
Cyklů	32
Kódování	0 0 1 OPR OPR OPR 0 0 0 0

Poznámka. Spočte inverzní tangens funkci (atan) zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR0. Zdrojový úhel musí být zadán v radiánech. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přenosti a vlajka Precision Exception (PE) je nastavena. V takovém případě zůstává cílový operand AR0 nezměněn.

TAN

Funkce	Tangens
Slova	1
Cyklů	64
Kódování	0 0 0 OPR OPR OPR 0 0 0 0

Poznámka. Spočte funkci tangens (tan) zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR0. Vedlejším účinkem instrukce je znehodnocení obsahu AR1. Zdrojový úhel musí být zadán v radiánech. Tento proces je dokončen ve dvou krocích, neboť funkci tan nelze spočítat přímo s užitím CORDIC mechaniky. Nejprve je vykonána instrukce SINCOS, která zanechá hodnotu funkce sinus v AR1. Následně je prostřednictvím DIV instrukce proveden podíl AR1 a AR0, jehož výsledek je uložen do AR0. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přenosti z důvodu nekonvergence algoritmu a vlajka Precision Exception (PE) je nastavena. V takovém případě oba z cílových operandů AR0 a AR1 zůstavají nezměněny.

SINHCOSH

Funkce	Hyperbolický Sinus a Cosinus
Slova	1
Cyklů	32
Kódování	0 1 1 OPR OPR OPR 0 0 0 0

Poznámka. Spočte funkce hyperbolického sinu a cosinu (sinh, cosh) zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR1 a AR0, respektivně. Zdrojový úhel musí být zadán v radiánech a ležet v intervalu $[-1.74; +1.74]$. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přenosti z důvodu nekonvergence algoritmu a vlajka Precision Exception (PE) je nastavena. V takovém případě zůstavají cílové operandy AR1 a AR2 nezměněny.

ATANH

Funkce	Inverzní hyperbolický tangens
Slova	1
Cyklů	32
Kódování	0 1 0 OPR OPR OPR 0 0 0 0

Poznámka. Spočte inverzní hyperbolickou tangens funkci (atanh) zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR0. Zdrojový úhel musí být zadán v radiánech a ležet v intervalu $[-0.80964; +0.80694]$. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přenosti z důvodu nekonvergence algoritmu a vlajka Precision Exception (PE) je nastavena. V takovém případě zůstává cílový operand AR0 nezměněn.

TANH

Funkce	Hyperbolický tangens
Slova	1
Cyklů	64
Kódování	1 0 1 OPR OPR OPR 0 0 0 0

Poznámka. Spočte hyperbolickou funkci tangens (tan) zdrojového operandu specifikovaného pomocí OPR a výsledek uloží do AR0. Vedlejším účinkem instrukce je znehodnocení obsahu AR1. Zdrojový úhel musí být zadán v radiánech. Tento proces je dokončen ve dvou krocích, neboť funkci tan nelze spočítat přímo s užitím CORDIC mechaniky. Nejprve je vykonána instrukce SINHCOSH, která zanechá hodnotu funkce sinus v AR1. Následně je prostřednictvím DIV instrukce proveden podíl AR1 a AR0, jehož výsledek je uložen do AR0. Pokud je zdrojový operand mimo rozsah, pak je detekována ztráta přesnosti z důvodu nekonvergence algoritmu a vlajka Precision

Exception (PE) je nastavena. V takovém případě oba z cílových operandů AR0 a AR1 zůstavají nezměněny.

LN										
Funkce	Přirozený logaritmus									
Slova	1									
Cyklů	96									
Kódování	1	1	1	OPR	OPR	OPR	0	0	0	0

Poznámka. Spočte přirozený logaritmus čísla specifikovaného pomocí OPR a uloží ho do AR0. Tato operace je třífázový proces, neboť $\ln(w)$ nelze spočítat přímo s užitím CORDIC mechaniky. Místo toho je využito následující identity

$$\ln(w) = 2\,\mathrm{atanh}\left(\frac{w-1}{w+1}\right) \tag{3.31}$$

Nejprve je vykonána DIV instrukce, která spočte podíl $w+1$ a $w-1$. Užití $w+1$ a $w-1$ zajistí, že reálná část argumentu je vždycky menší než část imaginární a hlavně, že realná část není nikdy rovna imaginární části. V opačném případě se výstup funkce atanh(w) blíží nekonečnu a výpočet nebude konvergovat. Následně je vykonána instrukce ATANH, jejímž výsledkem je inverzní hyperbolická tangenta. Konečně je výsledek zvojnásoben přičtením výsledku sám k sobě. Výsledek tedy je $\ln(w)$. Počítač může spočítat i logaritmy libovolného základu b a to s pomocí známé identity,

$$\log_b(w) = \frac{\ln(w)}{\ln(b)} \tag{3.32}$$

která vyžaduje výpočet pouze jediného dodatečného logaritmu a vykonání DIV instrukce pro požadovaný výsledek.

ADD										
Funkce	Součet									
Slova	1									
Cyklů	2									
Kódování	OPR	OPR	OPR	OPR	OPR	OPR	0	0	0	1

Poznámka. Přičte hodnotu zdrojového operandu specifikovaném OPR bity 6–4 k hodnotě cílového operandu specifikovaném OPR bity 9–5. Výsledek součtu je pak uložen do cílového operandu. Pokus o sečtení operandů rozdílných měřítek vyústí ve ztrátu přesnosti, neboť jeden z operandů bude automaticky přepočten do společné přesnosti 10^{-6}. Tato akce rovněž nastaví Precision Exception (PE) vlajku. Dále pokud při součtu dojde k přetečení, je automaticky nastavena Overflow Exception (OE) vlajka.

COM

Funkce	Převrácení znaménka
Slova	1
Cyklů	2

Kódování

X	0	0	OPR	OPR	OPR	0	0	1	0

Poznámka. Aplikuje dvojkový doplněk (*two's complement*) na operand specifikovaný pomocí OPR a výsledek zapíše zpět do stejného umístění. Tato operace tedy převrací znaménko daného registru. Odpovídajícím způsobem je rovněž nastavena Negative Flag (NF) stavová vlajka.

MPY

Funkce	Součin
Slova	1
Cyklů	17

Kódování

OPR	OPR	OPR	OPR	OPR	OPR	0	1	0	1

Poznámka. Vynásobí operandy specifikované OPR bity 6–4 a 9–5. Pokus o násobení operandů rozdílních měřítek vyústí ve ztrátu přesnosti, neboť jeden z operandů bude automaticky přepočten do společné přesnosti 10^{-6}. Tato akce rovněž nastaví Precision Exception (PE) vlajku. Výsledek součinu je 66-bitové číslo, proto má instrukce dvě write-back fáze. Spodních 33 bitů je zapsáno do AR0, horních 33 bitů do AR1.

DIV

Funkce	Podíl
Slova	1
Cyklů	32

Kódování

OPR	OPR	OPR0	OPR	OPR	OPR	0	1	0	0

Poznámka. Vydělí cílový operand specifikovaný OPR bity 9–7 zdrojovým operandem specifikovaným OPR bity 6–4. Vlajka Zero Divide Exception (ZE) je nastavena při pokusu o dělení nulou. Dělení je realizováno prostřednictvím CORDIC algoritmu v lineárních souřadnicích.

CMP

Funkce	Porovnání
Slov	1
Cyklů	2

Kódování

OPR	OPR	OPR	OPR	OPR	OPR	0	1	1	0

Poznámka. CMP porovná prostřednictvím rozdílu dva operandy OPR1 a OPR2 a dle výsledku nastaví příslušné stavové vlajky ZF (Zero Flag) a NF (Negative Flag) v SR registru. Výsledek samotného rozdílu je ignorován a zdrojové operandy zůstavají nezměněny.

LD										
Funkce	Datový přenos									
Slova	4									
Cyklů	Variabilní									
Kódování	IMM	IMM	IMM	IMM	IMM	OPR	0	1	1	1

Poznámka. Nahraje do Transfer registru *okamžitá* (immediate) nebo *nepřímá* (indirect) data. Pokud je nastaven OPR bit, pomocný registr Aux B je použit jako ukazatel do datové paměti, ze které jsou data vyčtena, v opačném případě je zvolena okamžitá adresa. V obou případech jsou zdrojová data rozdělena do tří 12-bitových slov a uložena v *little-endian* formátu. Původní[9] 10-bitová LD instrukce je rovněž zapsána do FXP koprocesoru před vlastním zapsáním dat.

Pokud počítač zapíše příliš mnoho dat a dojde k přetečení fronty FXP, vlajka Bus Exception (BE) je nastavena. Tento *extra kód* je zpracován jak počítačem tak FXP.

ST										
Funkce	Datový přenos									
Slova	4									
Cyklů	Variabilní									
Kódování	IMM	IMM	IMM	IMM	IMM	OPR	1	0	0	0

Poznámka. Zapíše obsah Transfer Registru do výstupní sběrnice. ST je blokovací instrukce a tento extra kód je zpracován jak počítačem tak koprocesorem FXP. Příslušná výkonná jednotka pozastaví veškeré vykonávání dalších instrukcí a vyčká na *FXP Output Acknowledge* signál *fxp_ inpt_ sta* před začátkem samotného datového přesunu. Maximální povolený čas blokování (*timeout*) je 500 us, po kterém příslušná jednotka automaticky aktivuje Computer Alarm (CA) signál a vstoupí do bezpečného režimu (s31).

V závislosti na OPR bitu jsou dostupné dva adresovací režimy:

OPR	**Transfer Mode**
0	**Immediate.** Data vyčtená z výstupní FXP sběrnice budou zapsána na okamžitou adresu v datové paměti 0–31.
1	**Indirect.** Data vyčtená z výstupní FXP sběrnice budou zapsána na adresu danou ukazatelem registru Aux B.

Poznámka. Všechny FXP instrukce jsou z definice dokončeny mnohem

[9] Originální programové slovo je zbaveno všech XTR bitů a je respektuje formát definovaný v sekci 3.9.

dříve než je nastavený blokovací limit 500 us. Příslušná výkonná jednotka má tedy vždy dostatek času dokončit jakýkoliv ST datový přenos.

LDC

Funkce	Konstatní datový přesun
Slova	1
Cyklů	3
Kódování	X IMM IMM OPR OPR OPR 1 0 0 1

Poznámka. Uloží do jednoho z aritmetických registrů specifikovaném pomocí bitu OPR.0 příslušnou konstantu. Tímto způsobem je eliminována potřeba užití komplikovanějších LD a TRA instrukcí pro přenos základních a hojně uživaných numerických hodnot. V závislosti na hodnotě OPR.2–0 bitu bude uložena jedna z následujích konstant:

OPR	**Konstanta**	**Přesnost**
00	π	3.141592653
01	e	2.718281828
10	1	1.000000000
11	0	0.000000000

Z kódovacího schématu je zřejmé, že LDC lze aplikovat pouze na AR0 a AR1.

TRA

Funkce	Datový přesun
Slova	1
Cyklů	2
Kódování	MOD X X OPR OPR OPR 1 0 1 0

Poznámka. V závislosti na MOD operandu buď zkopíruje obsah Transfer registru do jednoho z Aritmetických registrů specifikovaných OPR operandem, nebo naopak zkopíruje obsah zdrojového Aritmetického registru do Transfer registru.

MOD	**Direction**
0	Transfer Register → Arithmetical Register
1	Arithmetical Register → Transfer Register

Do samotného Transfer registru lze zapsat data prostřednictvím LD instrukce. Užitím instrukce ST lze pak zapsat obsah Transfer registru na výstupní FXP sběrnici.

MOV

Funkce	Registrový přesun
Slova	1
Cyklů	2
Kódování	OPR OPR OPR OPR OPR OPR 1 1 1 0

Poznámka. Zkopíruje obsah zdrojového registru specifikovaného OPR bity 6–4 do cílového registru specifikovaném OPR bity 9–7. Zdrojový operand zůstává touto operací nezměněn. Vzájemně přesouvat lze pouze aritmetické registry AR_i. Pro přesun mezi aritmetickými a transfer registry slouží instrukce TRA, popsaná rovněž v tomto oddíle.

CNDX

Funkce	Přepínač nepřímého přístupu
Slova	1
Cyklů	2
Kódování	0 X X X X OPR 1 1 1 1

Poznámka. Přepíná mezi nepřímým (*indirect*) přístupem do datové a programové paměti a to dle hodnoty OPR operandu. Pokud OPR = 0 pak má registr B funkci ukazatele do datové datové paměti, jinak do programové. Pokus o zápis do programové paměti[10], tedy

```
cndx prog
sto  b
```

není architekturou podporován a vyústí v nedefinovné chování počítače (*undefined behaviour*).

XCHG

Funkce	Výměna registrů
Slova	1
Cykly	4
Kódování	1 X X X X X 1 1 1 1

Poznámka. Vzájemně vymění obsahy registrů A a B. Tento extra kód nepřísluší FXP koprocesoru a je vykonán pouze danou výkonou jednotkou.

RND

Funkce	Náhodné číslo
Slova	1
Cyklů	2
Kódování	X X X OPR OPR OPR 0 0 1 1

Poznámka. Vyčte pseudo-náhodné číslo z volně běžícího[11] (*free-running*) LFSR[9] registru a uloží je do cílového registru specifikovaném pomocí OPR. Výsledné pseudo-náhodné číslo pak leží v intervalu $[1; 65535]$.

[10] Programová pamět je pouze pro čtení.

[11] Doba vyčtení dat z registru je tudíž náhodná.

ABS

Funkce	Absolutní hodnota
Slova	1
Cyklů	32
Kódování	0 1 X OPR OPR OPR 1 0 1 1

Poznámka. Spočte absolutní hodnotu vektoru $\boldsymbol{v} = (x, y)$ spolu s úhlem mezi vektorem $\boldsymbol{v}$ a $(0, 0)$. Zdrojový vektor $\boldsymbol{v}$ je specifikován pomocí OPR bity 4–6. Absolutní hodnota je uložena do registru AR0 a úhel do AR1.

SQRT

Funkce	Odmocnina
Slova	1
Cyklů	32
Kódování	1 1 0 OPR OPR OPR 0 0 0 0

Poznámka. Spočte odmocninu ze zdrojového operandu specifikovaného pomocí OPR a výsledek zapíše zpět do stejného umístění. Vstupní argument musí být nejprve zobrazen na hodnotu ležící v intervalu $[0.5, 2)$ a po kalkulaci zobrazen zpět na hodnotu původní. Vstup musí být nezáporný, jinak je nastavena vlaka Precision Exception (PE). Registr AR0 je znehodnocen, neboť v průběhu výpočtu je automaticky nastaven na hodnotu $1/K$ z důvodu kompenzace zisku rotace. Po dokoneční výpočtu bude AR0 obsahovat původní, nekompenzovanou hodnotu SQRT.

WAIT

Funkce	Vyčkání na FXP
Slova	1
Cyklů	Variabilní
Kódování	0 0 X X X X 1 0 1 1

Poznámka. WAIT pozastaví činnost příslušné výkonné jednotky počítače až do resetování SR.BSY bitu. Výpočetní jednotka vyčká v **blokovacím** stavu až do doby, kdy budou všechny iterativní procesy CORDICu a MPY jednotky dokončeny. Pokud je navíc nastaven bit Done Mask (DM) a Interrupt Enable Mask (IEM), pak jednotka okamžitě vstoupí do příslušného vektoru přerušení.

Na neiterativní instrukce, jako jsou například datové přesuny či součet, nemá WAIT žádný efekt, neboť SR.BSY je resetován mnohem dříve, než by mohl být počítačem analyzován. Dále, ST a LD jsou sami o sobě *blokovacími* instrukcemi, což opět nechává WAIT instrukci bez efektu.

LCSW

Funkce	Přesun kontrolního a stavového slova
Slova	3
Cyklů	Variabilní
Kódování	IMM IMM IMM IMM IMM IMM 1 1 0 0

Poznámka. Vyčte *stavové* (status) a *kontrolní* (control) slovo z datové paměti a uloží je do SR a CR registrů, respektivně. LCSW je principiálně podobné LD instrukci, neboť nejprve je zapsán extra kód instrukce, zatímco data jsou rozdělena do 12-bitových slov a zapsána separátně. Kontrolní slovo je zapsáno první, stavové jako druhé.

SCSW

Funkce	Přesun kontrolního a stavového slova
Slova	3
Cyklů	Variabilní
Kódování	IMM IMM IMM IMM IMM IMM 1 1 0 1

Poznámka. Uloží *stavové* (status) a *kontrolní* (control) slovo do datové paměti. SCSW je principiálně podobné ST instrukci z důvodu shodné synchronizace datového přenosu. Kontrolní slovo je v paměti uloženo na bitových pozicích 23–12, stavové na 11–0. Pořadí uložených bitů následuje identický vzor jako je tomu v definici samotných CR a SR registrů.

SCALE

Funkce	Změna měřítka
Slova	1
Cyklů	5
Kódování	1 0 IMM OPR OPR OPR 1 0 1 1

Poznámka. Provede změnu měřítka numerické hodnoty v umístění specifikované OPR operandem v závislosti na IMM bitu.

IMM	Scale	Range
0	10^{-9}	± 4.294967295
1	10^{-6}	± 4294.967295

Koprocesor implementuje tyto operace prostřednictvím rychlých *decimálních* posunů, jejichž princip byl rozebrán v oddílu (3.7.2). SCALE instrukce mění CONF bity zdrojového operandu.

LDS

Funkce	Levý desetinný posun
Slova	1
Cyklů	3
Kódování	X 0 1 OPR OPR OPR 0 0 1 0

Poznámka. Provede levý desetinný posun operandu specifikovaného pomocí OPR a výsledek zapíše zpět do stejného umístění. LDS tedy vynásobí daný operand deseti. Instrukce nemění CONF bity operandu. Pro změnu měřítka užijte SCALE.

RDS

Funkce	Pravý desetinný posun
Slova	1
Cyklů	5
Kódování	X 1 0 OPR OPR OPR 0 0 1 0

Poznámka. Provede pravý desetinný posun operandu specifikovaného pomocí OPR a výsledek zapíše zpět do stejného umístění. RDS tedy vydělí daný operand deseti. Instrukce nemění CONF bity operandu. Pro změnu měřítka užijte SCALE.

RETI

Funkce	Návrat z přerušení
Slova	1
Cyklů	5
Kódování	1 1 X X X X 1 0 1 1

Poznámka. Provede návrat z procedury obsluhy přerušení ISR. Instrukce obnoví FLG, programový segment CMS a programový čítač PC.

SIW

Funkce	Uložení slova přerušení
Slov	1
Cyklů	3
Kódování	0 1 0 X X X 1 1 1 1

Poznámka. Zapíše obsah A, B do registrů přerušení IFR a IER, respektivně. Instrukci SIW lze použít k resetování zdrojů přerušení na konci ISR procedury.

LIW	
Funkce	Načtení slova přerušení
Slov	1
Cyklů	3
Kódování	1 1 0 X X X 1 1 1 1

Poznámka. Zkopíruje obsah registrů přerušení IFR a IER do registru A a B, respektivně. LIW lze použít ke zjištění zdroje hlavního přerušení na začátku ISR procedury.

3.9.5 Instrukční alias

K vyřešení mnemonických konfliktů některých FXP extra kódů a běžných instrukcí výkonných jednotek byly definovány instrukční *aliasy*, nebo-li alternativní mnemonické označení. Překladač pak může tyto využít, čímž se zcela eliminuje nutnost podrobněji zkoumat význam dané mnemoniky. Například následující XTR instrukce

```
MOV AR5, AR0 ; Copy AR0 to AR5, FXP Extra Code (XTR)
```

má zcela jiný význam než instrukce,

```
MOV A, B     ; Copy Auxiliary B to Accumulator A
```

která náleží výpočetní jednotce počítače. Pokud bychom měli v příkladech pokračovat, pak výraz

```
MOV A, AR0   ; Copy AR0 to Accumulator A, no such instruction
```

odpovídá neplatné operace, neboť přímý přesun dat mezi registry výkonných jednotek počítače a FXP registry není architekturou podporován.

Překladač tedy může buď odvodit správný instrukční typ v závislosti na bližší inspekci operandů nebo využít následujících alternatvních jmenných definic.

Extra Code	**Alias**
ADD	AD
MPY	MP
DIV	DV
MOV	MV

Alternativní mnemonické definice tudíž výrazně zjednodušují proces tokenizace vstupního souboru.

Kapitola 4

Integrované periférie

4.1 VGA video

V této podkapitole bude rozebrán VGA video subsystém počítače. Implementované VGA video zahrnuje textovou video paměť RAM a ROM paměť bitové masky znakové palety. K VGA lze přistupovat pomocí tří paměťově mapovaných registrů a to **Display Control Register** (DCR), **Cursor Display Register** (CDR) a **Character Register** (CHAR).

4.1.1 VGA časování

Počítač implementuje pouze jediný video režim SVGA 800 x 600 @ 60 Hz a to z důvodu příhodné frekvence pixelového hodinového signálu (*pixel clock*). Kmitočet 40 MHz lze snadno a přesně syntetizovat s užitím celočíselného PLL poměru, jelikož systémový hodinový signál je 100 MHz. Dělicí faktor 4/10 je tedy vhodnější než například faktor 1007/4000 nutný pro standardní VGA režim 640 x 480 @ 60 Hz. Právě tento poměr by musel být na úkor přesnosti výstupního kmitočtu[1] zaokrouhlen na 1/4.

Signály definované standardem VGA jsou shrnuty v tabulce níže. Časování bylo vypočteno pro video režim 800 x 600 @ 60 Hz. Signály horizontální (*Sync pulse* HSYNC) a vertikální (*Sync pulse* VSYNC) synchronizace slouží k odlišení doby aktivního videa a zatemňovacích pulsů. Po naskenování celého řádku musel být elektronový svazek CRT obrazovky vypnut a elektronové dělo přesměrováno na následující řádek (*retract*). Video signál nesmí být v tuto chvílí vysílán. Video ovladač počítače proto nastaví RGB signály do log.0 úrovně. Stejné pravidlo platí pro vertikální zatemňovací puls, kdy byl svazek elektronů přesměrován z posledního zpět na první

[1] Video režim 640 x 480 @ 60 Hz vyžaduje frekvenci pixelového hodinového signálu 25.175 MHz. Užití zaokrouhleného faktoru 1/4 tak představuje chybu 0.175 MHz. V záležitosti na toleranci daného displeje, který se liší s výrobním procesem, by mohlo selhat zachycení (*lock*) synchronizačníhch signálů.

řádek. Nezávisle na implementované zobrazovací technologii musí každé VGA kompatibilní zařízení následovat tento popsaný formát.

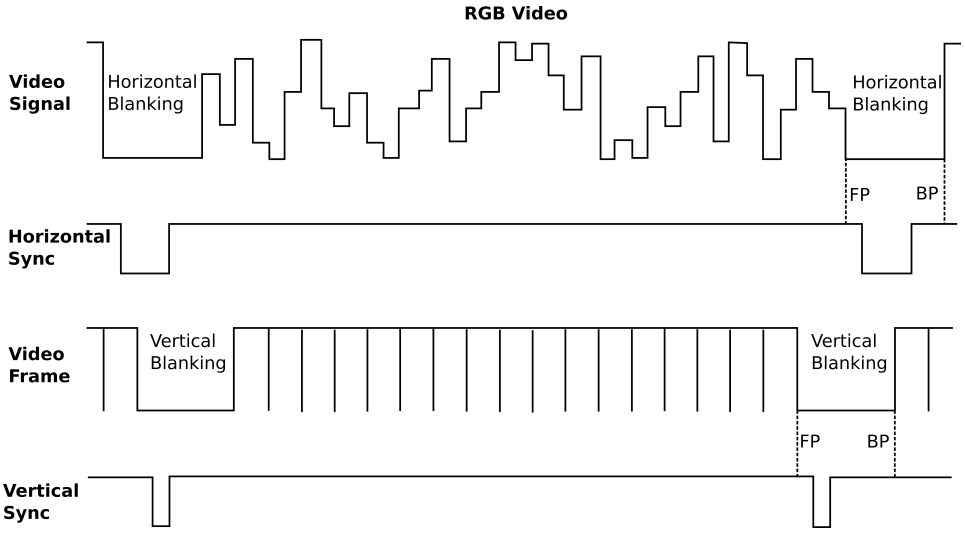
Fyzicky má VGA ovladač tři výstupní signály, které jsou propagovány až do nevrchnějšího modulu návrhu:

- Puls horizontální synchronizace.
- Puls vertikální synchornizace.
- Video signál rozdělený do tří barevných RGB komponent – červené *Red*, zelené *Green* a modré *Blue*.

Screen refresh rate [Hz]		**Vertical refresh [kHz]**		**Pixel freq. [MHz]**
GENERAL TIMING				
60		37.878787878788		40
Scanline	**Pixels**	**Time [µs]**	**Frame**	**Lines**	**Time [ms]**
HORIZONTAL			**VERTICAL**		
Visible area	800	20	Visible area	600	15.84
Front porch	40	1	Front porch	1	0.0264
Sync pulse	128	3.2	Sync pulse	4	0.1056
Back porch	88	2.2	Back porch	23	0.6072
Whole line	1056	26.4	Whole frame	628	16.5792

- Polarita horizontálního synchronizačního pulsu je kladná.
- Polarita vertikálního synchronizačního pulsu je kladná.

Pro názornost je VGA časování vyobrazeno na časovacím schématu níže:



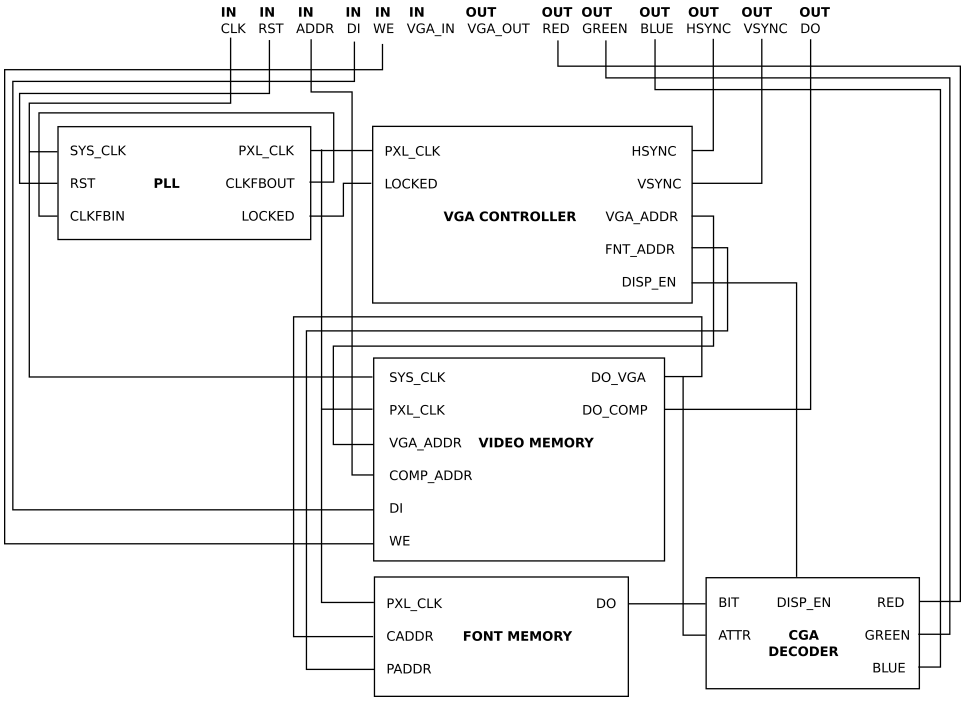
Obrázek 4.1: Video VGA časování.

4.1.2 VGA systém

Celý VGA subsystém sestává z bloků ovladače videa **VGA Controller**, který generuje synchronizační a *display enable* signály spolu se signály adres video paměti a paměti znakové palety, video paměti **Video Memory** o kapacitě 4000 znaků a konečně paměti znakové palety **Font Memory**, která obsahuje odpovídající bitovou masku (*bitmap*) pro každý z 64 dostupných znaků.

Systém používá bitovou masku o rozměrech 8x15 pro každý tisknutelný i netisknutelný znak. Pokud je příslušný bit v masce nastaven, popředí (*foreground*) znaku je vytištěno, v opačném případě je vytištěno pozadí (*background*). Celá obrazovka je tudíž rozdělena přesně do 40 řádků a 100 sloupců. Ovladač videa snímá který z pixelů je v daný okamžik viditelný a provádí kontinualní znakové scanovaní. Každé pixelové pozici je přiřazena adresa ve video paměti, ze které jsou současně čteny tisknutelné znaky. Tisknutelný znak je spolu se spočtenou bitmapovou adresou zapsán na vstupy paměti znakovové palety, ze které je pak vždy vybrán jeden konkréntí bit k zobrazení.

Poznámka. Zobrazení pixelu na adresu video paměti není unikátní, protože video paměť je textová a obsahuje 4000 záznámu oproti 480 000 viditelných pixelů. Namísto toho přísluší jedné paměťové pozici pole o velikosti 8x15 pixelů – tedy právě jeden znak.



Obrázek 4.2: Video VGA schéma.

- Celý proces znakového scanování a bitmap překladu je realizován během jediného hodinového cyklu.
- Sysém používá 4-bitovou barevnou CGA paletu, která je pevná a nelze ji změnit pomocí uživatelského softwaru.

- Adresování video a znakové paměti musí být synchronizováno s VSYNC signálem a proto je po startu počítače první snímek typicky vynechán z důvodu vyčkání na závěs VSYNC signálu.
- Pokud dojde ke strátě závěsu PLL LOCKED[2] signálu, všechny pixelové čítače musí být drženy v resetovacím stavu až do doby, kdy je PLL schopno opět obnovit činnost zpětné vazby. V tuto chvíli musí video ovladač opět projít inicializačním stavem a zajistit synchronizaci paměťového adresování s VSYNC signálem.

4.1.3 Video paměť

Tisknutené znaky jsou uloženy v 4096 × 10–bitové paměti[3]. Paměťově pozice 0–21 jsou přednastaveny a obsahují řetězec "HELLO, COMPUTER READY.". Toto opatření slouží jako jednoduchý vizuální indikátor správné funkčnosti video ovladače hned po připojení VGA kompatibilního displeje. Jinak však nejsou paměťové pozice 0–21 ničím jedinečné a lze je přepsat regulérními uživatelskými znaky.

Následuje schéma paměťového slova video paměti.

MSB						**LSB**
11	10	9	6	5	...	0
RSVD	RSVD	ATTR	ATTR	CHR	...	CHR

- **Reserved (RSVD).** Reservovaný bit, výhozí nula.
- **Attribute (ATTR).** Znakový atribut, CGA barva popředí.
- **Character (CHR).** Kód znaku k vytištění.

Poznámka. V současné implementaci je pro jednoduchost vynucena barva pozadí černá (0x00).

VGA standard vyhrazuje 3 bity pro červenou a zelenou barvu a 2 bity pro barvu modrou. Užitá 4-bitová barevná paleta zahrnuje 16 standardních CGA barev, jak je vyobrazeno v tabulce níže.

Color Attribute	**RGB DAC Code**	**Color**
0x0	0x00	CGA_BLACK
0x1	0x02	CGA_BLUE
0x2	0x14	CGA_GREEN
0x3	0x16	CGA_CYAN

[2]LOCKED je asynchronní signál.

[3]Ve skutečnosti je navržená pamět rozměrů 4096×12 bitů, vzhledem k tomu, že nejvyšší dva MSB bity jsou rezervovány a zapisovány vždy jako nula, syntetizační nástroj odvodí paměť o velikosti 4096 × 10 bitů.

0x4	0xA0	CGA_RED
0x5	0xA2	CGA_MAGENTA
0x6	0xA8	CGA_BROWN
0x7	0xB6	CGA_GRAY
0x8	0x49	CGA_DARK_GRAY
0x9	0x4B	CGA_BRIGHT_BLUE
0xA	0x5D	CGA_BRIGHT_GREEN
0xB	0x5F	CGA_BRIGHT_CYAN
0xC	0xE9	CGA_BRIGHT_RED
0xD	0xEB	CGA_BRIGHT_MAGENTA
0xE	0xFD	CGA_YELLOW
0xF	0xFF	CGA_WHITE

CGA barevná paleta má pouze 4 úrovně a to 0x00, 0x55, 0xAA a 0xFF. Vydělením těchto úrovní hodnotou 0xFF tudíž získáme analogové hodnoty s krokem 1/3, tj. 0, 1/3, 2/3 a 1.

- Mapování CGA barev: $00 \mapsto 0, 55 \mapsto 1, \mathrm{AA} \mapsto 2, \mathrm{FF} \mapsto 3$

Video paměť je dvouportová (*dualport*) paměť řízená dvěma rozdílnými hodinovými signály (*dual clock*). Paměť je průběžně čtena na vzestupné hraně pixelového hodinového signálu (40 MHz), zatímco výkonné jednotky do ní mohou zapisovat na vzestupné hraně hlavního hodinového signálu (100 MHz). Ve video systému jsou tudíž přítomny dvě hodinové domény.

- VGA ovladač nemůže zapisovat do video paměti.
- Počítač[4] nemůže číst z video paměti.

Pro složitější manipulace s videem je nutné zavést a udržovat *double buffer*, aby měl softwarový ovladač přehled o viditelnosti daných znaků na obrazovce. Double buffer má shodnou velikost jako video paměť a lze jej číst i zapisovat, protože je uložen v systémové paměti. Veškeré vykreslovací změny jsou provedeny nejprve v double bufferu, jehož aktivní část je následně nakopírována do video paměti. Tímto způsobem lze plynule implementovat skrolování, textové animace a další efekty.

Poznámka. Zavedení softwarového double bufferu není v současné implementaci možné z prostého důvodu nedostatečné kapacity datové paměti, která je omezena na 2048 12-bitových slov (3 kB). Tento neduh lze poměrně jednoduše odstranit zavedením segmentace datové paměti, ke kterému je od samého začátku i uzpůsobeno kódování instrukcí. Podrobněji je toto téma rozvedno v oddílu závěru práce[12].

[4] Některé architektury, jako například x86, principiálně umožňují číst data z video paměti, neboť samotná video paměť je mapována v adresovém prostoru systémové paměti. I v toto případě však přímé čtení paměti není doporučováno, protože je *velmi* pomalé v porovnání s pouhým zapisováním.

4.1.4 Znaková paleta

Systémový 8×15 bitmap je uložen v 64×120 bitové ROM paměti a vzhledem velmi připomíná tradiční VGA BIOS font.

Poznámka. Z důvodu omezených zdrojů byla zvolena redukovaná znaková paleta namísto E/ASCII. Kódování zahrnuje 62 tisknutelných znaků sestávajících zejména z velkých tiskacích písmen a číslic, včetně mezery SPACE a 2 *netisknutelných* kontrolních znaků, NULL (Null Terminated String) a CRLF (Carriage Return and Line Feed).

	.0	**.1**	**.2**	**.3**	**.4**	**.5**	**.6**	**.7**	**.8**	**.9**	**.A**	**.B**	**.C**	**.D**	**.E**	**.F**
0.	NULL	CRLF	SPACE	#	!	%	&	’	(	)	*	+	,	-	.	/
1.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
2.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_

Tabulka 4.2: Znaková paleta

4.1.5 VGA rozhraní

Video systém je uživatelským softwarem řízen prostřednictvým tří registrů – DCR, CDR a CHAR. DCR obsahuje především stavovou vlajku zavěšení PLL, která je prvním indikátorem korektní funkčnosti videa.

DCR : DISPLAY CONTROL REGISTER **Write: 0x7F0** **Read: 0x7F0**

MSB											**LSB**
RSV	RSV	RSV	RSV	RSV	RSV	RSV	RSV	RSV	UPDT	TEST	LOCK

LOCK	DCR.00	R	**Locked.** Indikuje stav zavěšení zpětné vazby fázového závěsu PLL. Při dobrém zavěšení jsou automaticky spuštěny synchronizační čítače a video signál je generován.
TEST	DCR.01	R/W	**Test.** Nastavením tohoto bitu lze spustit zkušební stránku sestávající z 16 vertikálních pruhů, kde každý pruh má právě jednu barvu z palety CGA. Tento režim potlačí výstupy z video paměti. Pro normální režim je tento bit resetován.
UPDT	DCR.02	W	**Update.** Toto je interní *write-enable* signál registrů DCR a CDR. Uživatelský zápis přes rozhranní MMIO je hardwarem ignorován.
RSVD	DCR.03-11	NI	**Reserved.** Čteno jako nula.

CDR : CURSOR DISPLAY REGISTER **Write: 0x7EF** **Read: None**

MSB											**LSB**
D	D	D	D	D	D	D	D	D	D	D	D

D CDR.00-11 W **Data.** Pozice lineárního kursoru 0–3999.

CHR : CHARACTER REGISTER **Write: 0x7EE** **Read: None**

MSB											**LSB**
RSVD	RSVD	ATT	ATT	ATT	ATT	CH	CH	CH	CH	CH	CH

CH	CHR.00-05	W	**Character.** Znak, který bude vytištěn na obrazovce. Přehled platných znakových kódu lze nalézt v (4.1.4).
ATT	CHR.06-09	W	**Attribute.** Barevný atribut znaku příslušný znaku, který bude vytištěn na obrazovce. Přehled platných barevných atributů lze nalézt v (4.1.3)
RSVD	CHR.10-11	NI	**Reserved.** Čteno jako nula.

Poznámka. Zápis do CHR registru *neinkrementuje* kursor v DCR.

4.2 PS/2 rozhraní

Počítač implementuje podmožinu PS/2 kompatibilního ovladače a to konkrétně klávesnici. Auxiliární zařízení[5] PS/2 jsou ignorována. Port ovladače má dva signální vstupy:

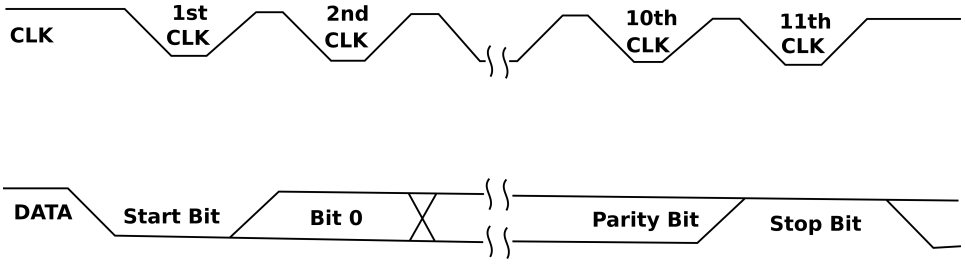
CLK	**Clock.**	Hodinový signál generovaný PS/2 kompatibilním zařízením, zde klávesnicí. Kmitočet oscilátoru je typicky 16.7 kHz, může však nabývat hodnot od 10 kHz do 16.7 kHz.
DATA	**Data.**	Datový signál, generovaný zařízením, které dle PS/2 protokolu právě zasílá data.

Datový rámec sestává ze sekvence 11 bitů, 1 start bit, 8 datových bitů, 1 paritní bit a 1 stop bit. Data jsou zapisována od LSB po MSB a jsou čtena na sestupné hraně hodinového signalu CLK.

Bity	Funkce
10	Stop bit, vždy log.1
9	Paritní bit, lichá parita
8–1	Datové bity v pořadí od MSB do LSB
0	Start bit, vždy log.0

[5] Auxiliárním zařízením je myšleno libovolné PS/2 kompatibilní zařízení se sériovým vstupem a výstupem.

Tabulka 4.5: Datový rámec PS/2 transakce
Níže je formát rámce znázorněn na časové ose.



Obrázek 4.3: Datový rámec PS/2 transakce
Počítač je vždy *bus master* a ovladač neumožňuje odesílat data. Sběrnice je neaktivní pokud jsou oba signály CLK a DATA ve vysoké úrovně. Přijímací sekvence je následující:

1. Začátek transakce je indikován stažením *hodinového* vodiče CLK do nízké úrovně.

2. První bit je START bit a musí být 0. Nyní zařízení odešle 8 datových bitů.

3. Počítač sekvenčně načte na každé sestupně hraně hodinového signálu CLK jeden DATA bit. Pokud je CLK inaktivní více než 100 µs (nejvyšší přípustná perioda protokolu PS/2), pak došlo k selhání hardwaru nebo byl z jiného důvodu přenos přerušen.

4. V desátem hodinovém cyklu je přijat paritní bit a v jedenáctem hodinovém cyklu STOP bit. Počítač nyní ověří (lichou) paritu a ověří ji s paritou přijatou v 10 hodinovém cyklu. Pokud se parita shoduje, je nastaveno přerušení IRQ.

5. CLK a DATA se vrátí do neaktivní, vysoké úrovně a systém je připravem k další transakci.

K detekci sestupné hrany je nejvýhodnější implementovat detektor hrany. Hodinový signál (10 kHz – 16.7 kHz) sběrnice CLK tak může být synchronizován s doménou hlavního hodinového signálu (100 MHz), čím efektivně zamezíme jevu CDC (*Clock Domain Crossing*), neboť v návrhu zůstane pouze jediný hodnový signál, a to systémový.

Poznámka. Signály CLK a DATA jsou řízeny výstupem s otevřeným kolektorem a vyžadují připojení *pull-up* rezistorů na 5 VDC napájení. Pull-up rezistory nejsou součástí klávesnice.

Kapitola 5

Softwarové nástroje

Vzhledem k narůstající komplexnosti celého návrhu bylo nutné implmentovat i některé podpůrné softwarové komponenty, a to především překladač jazyka symbolických adres (*assembler*) a emulátor celého počítače.

5.1 Překladač

Jazyk symbolických adres je programovací jazyk nízké úrovně tvořený symbolickou reprezentací jednotlivých instrukcí počítače. Překladač mapuje tyto pro lidského operátora snadno čitelné symbolické reprezentace (*mnemoniky*) instrukcí do jejich odpovídají numerické hodnoty. Jazyk se typicky skládá ze dvou komponent:

- **Instrukční mnemonika.** Každé ze strojových instrukcí je přiřazen unikátní řetězec tisknutelných znaků, které zároveň vystihují význam a účel dané instrukce. Podobně jsou přiřazena mnemonická označení i registrům či stavovým vlajkám. Rovněž každé paměťové pozici je možné přiřadit příhodné návestí (*label*), na které je pak možné symbolicky odkazovat operandy instrukcí. Tato funkce je velice důležitá, neboť zcela eliminuje nutnost po každé změně délky programu ručně přepočítat veškeré relativní paměťové odkazy.

- **Directivy překladače.** Tyto zahrnují veškeré příkazy, které nějakým způsobem ovlivňují vlastní funkci assembleru. Může jít o složitá makra, preprocesor konstatního matematického výrazu či ruční deklaraci dat a konstant.

Zobrazení mnenomického označení instrukce a její skutečné numerické hodnoty je často bijektivní a význam doslovný, explicitní – není nutné žádná další dedukce či analýza daných výroků zdrojového kódu. Ač designově jednodušší, jazyk symbolických adres je prakticky nepřenositelný na jiné platformy a je přísně limitován možnostmi dané architektury. Obecně rozlišujeme dva typy

překladačů podle počtu průchodů (*passes*) zdrojového kódu nutných k vytvoření objektového souboru:

- **One–pass.** Překlač projde zdrojový kód pouze jednou. Existence jakéholiv nedefinovaného konstruktu vyústí po dokončení operace v chybové hlášení a ukončení překladu.
- **Multi-pass.** Překlač projde zdrojový kód minimálně dvakrát. Při prvním průchodu je vytvořena tabulka všech symbolů s odpovídajícími hodnotami, které jsou následně použity dále v procesu překládání.

Typ překladače má zcela zásadní vliv na organizaci a styl psaní uživatelského softwaru, jak můžeme pozorovat na příkladu níže.

```
          B     .label1
.label1:  B     .label2
.label2:  B     .label2
```

Při pokusu o překlad výše uvedeného kódu ohlásí *one-pass* překladač chybu, jelikož v době překladu není známa hodnota symbolu *.label1*, zatímco *multi-pass* překladač dokončí překlad úspěšně. V prvním kroku jsou totiž analyzovány veškeré symboly a k nim spočteny odpovídající hodnoty. Při druhém průchodu zdrojového kódu je již hodnota *.label1* známa a překlad může být dokončen.

Poznámka. Naprosto totožně můžeme rozlišovat i vysokoúrovňové jazyky. Například jazyky C a C++ jsou *one-pass* a proto kódové schéma vyžaduje existenci tzv. předsunutých deklarací (*forward declarations*).

Zpracování vstupního souboru proběhne v následujících třech krocích:

1. **Lexing.** Vstupní soubor je přečten, zbaven všech bílých znaků (*whitespaces*) a komentářů, setříděn do *lexikálního* stromu a předán druhé fázi. Pokud je během tohoto procesu nalezena INCLUDE direktiva, je nutno první fázi opakovat znovu a již existujicí lexikální strom příslušně upravit. Po úspešném dokončení lexovací fáze je vytištěn výstupní soubor obsahující veškeré rozponané tokeny spolu s jejich významem. Jakékoliv IO selhání nebo chyba syntaxe zastaví proces překladu.
2. **Parsing.** Každý list lexikálního stromu reprezentuje *token*, který je analyzován parserem. Topologicky nejvýznamější token je pak použit jako index do tabulky funkcí (*call table*), kde jsou uloženy veškeré operace parseru, jako zakódování instrukce či evaluace makro výrazu. Parser musí projít lexikální strom dvakrát (*two-pass*). První průběh nahradí veškeré konstanty a výrazy skutečnými numerickými hodnotami a rovněž spočte hodnoty všech deklarací programových návěstí. Druhý průběh pak provede již samotné zakódování instrukcí. Jakákoliv sémantická chyba zastaví proces překladu.

3. **Output.** V závislosti na nastavení je vygenerován výstupní soubor v příslušném formátu. Pokud je zvolen VHDL výstup, čas a datum vytvoření je rovněž vytišněno do výstupního souboru k odlišení mezi jednotlivými verzemi zdrojových souborů. Jakékoliv IO selhání nebo chyba syntaxe zastaví proces překladu.

- Překladač nepotřebuje žádný linker, neboť všechny zdrojové INCLUDE soubory jsou přečteny rekurzivně v první fázi překladu.
- Každý soubor reprezentuje separátní a nezávislou překladovou jednotku.

5.1.1 Definice klíčových slov

Následuje krátký seznam klíčových slov direktiv překladače spolu s jednoduchým popisem. Ještě podotkněme, že direktivy, podobně jako instrukční mnemoniky, nerozlišují mezi velkými či malými písmeny (*case-insenstive*).

Direktiva	Symbol	Význam
DIRECTIVE KEYWORDS		
ORG	**Origin.**	Označuje začátek zdrojového kódu. Ke každé relativní adrese je přičtena hodnota operandu direktivy. Typická hodnota je ORG 0. Pokud je direktiva vynechána pak je výchozí offset roven nule.
INCLUDE	**Include File.**	Vloží zdrojový soubor. Cesta k souboru může být relativní i absolutní. Nekonečná rekurze (soubor vkládá sám sebe) je potlačena pouze na úrovni daného souboru. Překlač nesestavuje žádný strom vzájemné dependence a proto musí problému rekurze zamezit lidský operátor[1].
GENERATE	**Generate Output.**	Specifikuje výstupní formát překladače, kterým může výstup binární BINARY nebo vhdl *inlcude* soubor VHDL, viz příklad (5.5).

[1] Podobně je tomu například v C či C++, kde je doporučeno vkládání tzv. *header guard* maker, které zamezí zmíněné nekonečné rekurzi.

FILENAME	**Input Filename.**	Specifikuje název výstupního souboru. Cesta k souboru může být relativní i absolutní.
CONSTANT	**Declare Constant.**	Deklaruje numerickou konstantu názvu a hodnoty specifikované operandy.
DPW	**Declare Packed Word.**	Deklaruje znakový řetězec.
DW	**Declare Word.**	Deklaruje plné 16-bitové programové slovo v numerické reprezentaci.

Tabulka 5.2: Definice klíčových slov direktiv překladače

5.1.2 Atributy

Překladač rovněž principiálně podporuje tzv. *operandové atributy*. Operandový atribut je podobný například známým atributům z jazyka hardwarového popisu VHDL či lexikálně příbuzeného (a historicky nadřazeného) programovacího jazyka ADA. Obecně atribut umožňuje vyzískat specifické informace o daném objektu, jako je délka proměnné v bitech, současný stav konkuretního procesu a jiné.

V současné verzi jsou implementovány pouze následující dva atributy:

Atribut	Význam
ATTRIBUTE KEYWORDS	
HIGH	Vrátí hodnotu 4 nejvyšších bitů operandu programového návěstí.
LOW	Vrátí hodnotu 12 nejnižších bitů operandu programového návěstí.
OFFSET	Vrátí relativní adresu programového návěstí.

Tabulka 5.4: Definice atributů překladače

Atribut je vždy předcházen symbolem apostrofu ' a musí být platný v kontextu operandu na který je aplikován. Následující příklad ilustruje užití jediného atributu programového návěstí.

```
        mov a, .label'high
.label: mov b, .label'low
```

Registr A bude obsahovat nejvyšší čtyři bity adresy *.label*, tzn. hodnotu 0, zatímco registr B nejnižších dvanáct bitů adresy *.label*, tedy hodnotu 1.

Poznámka. Atributy programového návěstí jsou kritické pro pohodlné nahrávání počátečních PC adres do LOAD registrů počítače. Bez užití atributů by bylo nutné po každé změně délky programu adresy ručně přepočítat. Zavedením atributů je tento úkol automatizován. Atributy jsou podporovány

na *lexikální* úrovni překladače, což činí implementaci případných dalších atributů velmi jednoduchou.

5.1.3 Postfix notace

Překladač je schopen v čase překladu evaluaovat konstatní matematický výraz. Tradiční infix notace, kdy jsou matematické operátory umístěny mezi operandy je však pro strojové zpracování nevhodná. Vstupní výraz tedy musí být před vlastním výpočtem převeden do postfix notace, kdy jsou jednotlivé matematické operátory umístěny za operandy. Výpočet pak může být proveden s užitím *zásobníkového* výpočetní stroje (stack machine).

Infix konverze

Definujme nyní *infix* výraz jako řetězec složený s tokenů matematických operátorů součtu, rozdílu, součinu, podílu, dále symbolu levé a pravé závorky a konečně i samotných numerických operandů. Algoritmus popsaný níže pak provede konverzi *infix* notace do *postfix* notace.

1. Je vytvořen prázdný zásobník (*stack*)jako dočasné úložiště matematických operátorů. Dále je inicializována fronta (*queue*) pro výstupní řetězec.

2. Vstupní řetězec je nyní procházen zleva doprava. V závislosti na hodnotě tokenu je proveda jedna z následujících operací:

   Pokud je token operand, pak je zařazen na konec výstupní fronty.

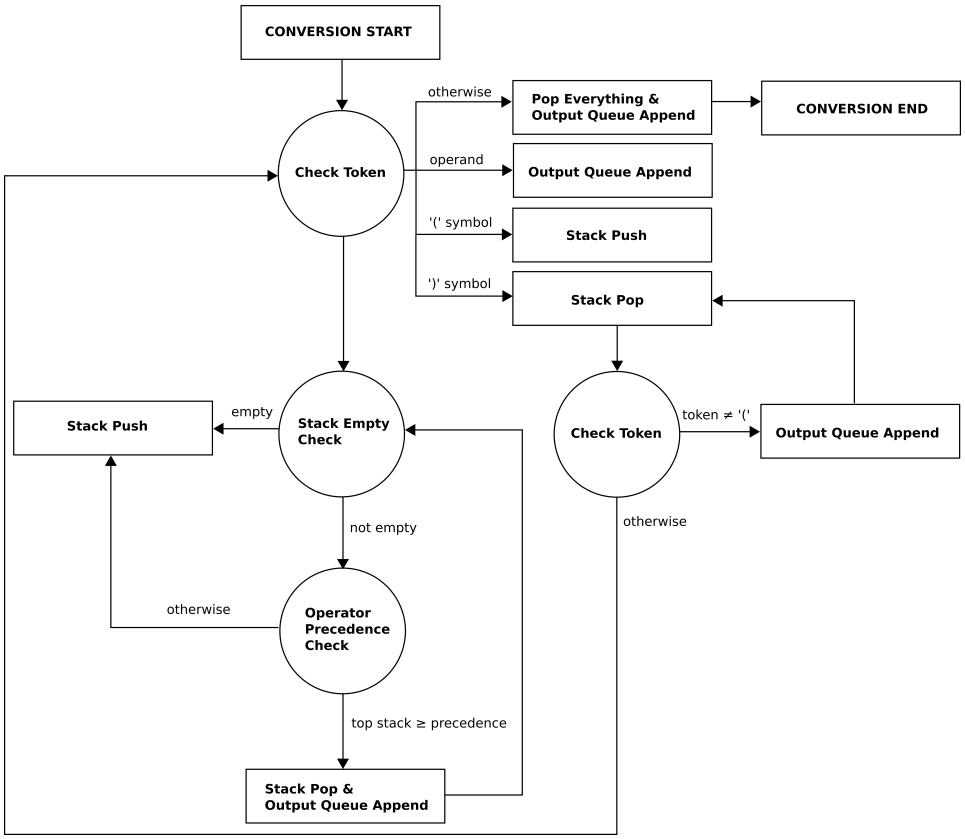
   Pokud je token symbol levé zárovky, pak je vložen (*pushed*) do zásobníku operátorů.

   Pokud je token symbol pravé závorky, pak je vyčten symbol z vrchu zásobníku (*poped*) a zařazen na konec fronty. Tento krok je opakován do té doby, kdy je vyčtený symbol roven symbolu pravé závorky.

   Pokud je token matematický operátor, pak je vložen do zásobníku operatorů. Tento krok předchází vyčtení všech operátorů vyšší nebo stejné precedence než je současně skenovaný token, a jejich postupné zařazení na konec výstupní fronty.

3. Krok 2 je opakován dokud není dosažen konec procházeného vstupního řetězce. Všechny zbývající operátory v zásobníku jsou nyní přečteny a uloženy na konec fronty. Převod z *infix* notace do *prefix* notace je tímto dokončen.

Pro názornost je výše uvedený algoritmus graficky znázorněn pomocí následujícího schématu:



Obrázek 5.1: Schéma infix–to–postfix konverze

Výpočet postfix vyrazu

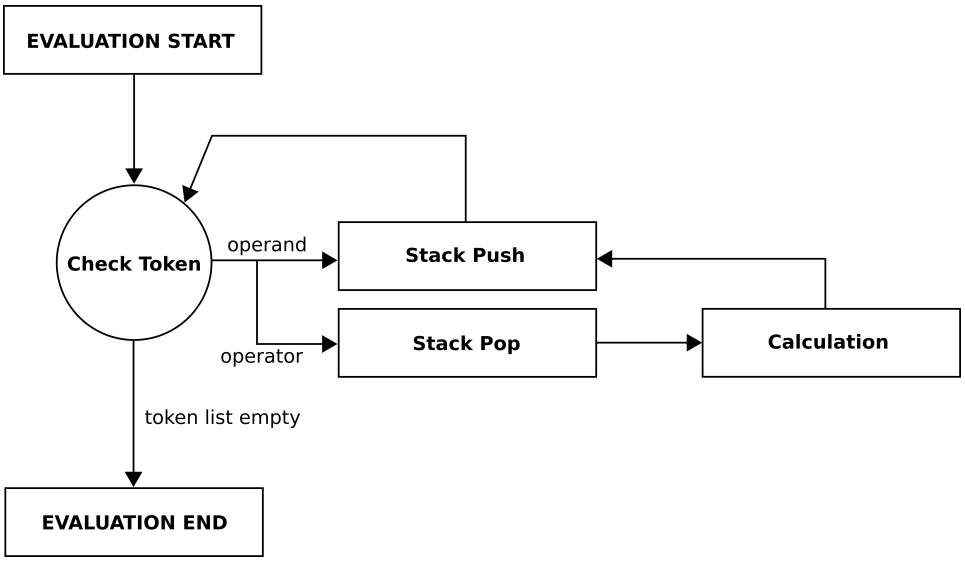
Definujme *postfix* výraz stejným způsobem jako v podkapitole 5.1.3. Níže popsaný algoritmus provede výpočet na retězci v *postfix* notaci.

1. Je vytvořen prázdný zásobník (*stack*).
2. Vstupní řetězec je nyní procházen zleva doprava. V závislosti na hodnotě tokenu je proveda jedna z následujicích operací:

   Pokud je token operand, pak je vložen do zásobníku (*pushed*).

   Pokud je token matematický operátor, pak jsou vyčteny dvě hodnoty (operandy) ze zásobníku (*poped*). Nyní je provedena příslušná aritmetická operace a výsledek uložen zpět do zásobníku.
3. Krok 2 je opakován dokud není dosažen konec procházeného vstupního řetězce. Výsledek kalkulace je pak k vyčtení na vrchu zásobníku.

Pro názornost je výše uvedený algoritmus graficky znázorněn pomocí následujícího schématu:



Obrázek 5.2: Algoritmus výpočtu postfix výrazu

Poznámka. Pořadí vyčtených operandů je kritické pro nekomutující matematické operátory, jako je například podíl. První hodnota vyčtená ze zásobníku tudíž odpovídá druhému operandu a druhé vyčtení operandu prvnímu.

5.1.4 Příklad

Níže je uveden tradiční *Hello World* příklad. Následující kód vytiskne řetězec "HELLO WORLD" na třetí viditelný řádek obrazovky.

Zdrojový kód (Vstup)

```
ORG 0
GENERATE VHDL
FILENAME hello_world.out
cms     #0
mov     b, .str'offset ; String address
mov     a, #200        ; 3th row, 0th col
sto     #40
call    .print_string
.halt:          b .halt                 ; Infinite loop

;*************************************************
;*          Print String From Program Memory     *
;*      Input: B string pointer, [40] cursor     *
;*      Output: None                             *
;*************************************************
; Video Registers
constant DCR_REG = #0x7F0
constant CDR_REG = #0x7EF
constant CHR_REG = #0x7EE
.print_string: lod      #40
sto     #CDR_REG  ; Set cursor
cndx    prog      ; Read program
.iter:          lod     b         ; Load 12 bits
sto     #41       ; Temporarily store the packed word
slr     #6        ; 1st packed character
and     #0x3F     ; Zero?
reteq
or      #0x1C0    ; Color attribute
sto     #CHR_REG  ; Print 1st char
lod     #40
add     #1
sto     #CDR_REG  ; Advance cursor
sto     #40

lod     #41
and     #0x3F     ; 2nd packed character
reteq             ; Zero?
```

```
or      #0x1C0
sto     #CHR_REG  ; Print 2nd char

lod     #40
add     #1
sto     #CDR_REG  ; Advance cursor
sto     #40

mov     a, b
add     #1        ; Increment pointer
mov     b, a
b       .iter
; Every string is zero-terminated automatically.
.str: dpw hello_world_str = "HELLO WORLD"
```

Výpis tokenizovaného vstupu první fáze

```
ASSEMBLER 1.00 \Debug\hello_world.txt CREATED: 14:19:37 | Thursday, April 5, 2017

Lexer Item 0 : (KEYWORD)ORG (VALUE)0
Lexer Item 1 : (KEYWORD)GENERATE (VALUE)VHDL
Lexer Item 2 : (KEYWORD)FILENAME (VALUE)hello_world.out
Lexer Item 3 : (INSTRUCTION)cms (OPERAND_IMM)#0
Lexer Item 4 : (INSTRUCTION)mov (OPERAND_REG)b (OPERAND_ADDR).str'offset
Lexer Item 5 : (INSTRUCTION)mov (OPERAND_REG)a (OPERAND_IMM)#200
Lexer Item 6 : (INSTRUCTION)sto (OPERAND_IMM)#40
Lexer Item 7 : (INSTRUCTION)call (OPERAND_ADDR).print_string
Lexer Item 8 : (LABEL).halt: (INSTRUCTION)b (OPERAND_ADDR).halt
Lexer Item 9 : (KEYWORD)constant (NAME)DCR_REG (VALUE)#0x7F0
Lexer Item 10 : (KEYWORD)constant (NAME)CDR_REG (VALUE)#0x7EF
Lexer Item 11 : (KEYWORD)constant (NAME)CHR_REG (VALUE)#0x7EE
Lexer Item 12 : (LABEL).print_string: (INSTRUCTION)lod (OPERAND_IMM)#40
Lexer Item 13 : (INSTRUCTION)sto (OPERAND_IMM)#CDR_REG
Lexer Item 14 : (INSTRUCTION)cndx (OPERAND_REG)prog
Lexer Item 15 : (LABEL).iter: (INSTRUCTION)lod (OPERAND_REG)b
Lexer Item 16 : (INSTRUCTION)sto (OPERAND_IMM)#41
Lexer Item 17 : (INSTRUCTION)slr (OPERAND_IMM)#6
Lexer Item 18 : (INSTRUCTION)and (OPERAND_IMM)#0x3F
Lexer Item 19 : (INSTRUCTION)reteq
Lexer Item 20 : (INSTRUCTION)or (OPERAND_IMM)#0x1C0
Lexer Item 21 : (INSTRUCTION)sto (OPERAND_IMM)#CHR_REG
Lexer Item 22 : (INSTRUCTION)lod (OPERAND_IMM)#40
Lexer Item 23 : (INSTRUCTION)add (OPERAND_IMM)#1
Lexer Item 24 : (INSTRUCTION)sto (OPERAND_IMM)#CDR_REG
Lexer Item 25 : (INSTRUCTION)sto (OPERAND_IMM)#40
Lexer Item 26 : (INSTRUCTION)lod (OPERAND_IMM)#41
Lexer Item 27 : (INSTRUCTION)and (OPERAND_IMM)#0x3F
Lexer Item 28 : (INSTRUCTION)reteq
Lexer Item 29 : (INSTRUCTION)or (OPERAND_IMM)#0x1C0
Lexer Item 30 : (INSTRUCTION)sto (OPERAND_IMM)#CHR_REG
Lexer Item 31 : (INSTRUCTION)lod (OPERAND_IMM)#40
Lexer Item 32 : (INSTRUCTION)add (OPERAND_IMM)#1
Lexer Item 33 : (INSTRUCTION)sto (OPERAND_IMM)#CDR_REG
Lexer Item 34 : (INSTRUCTION)sto (OPERAND_IMM)#40
Lexer Item 35 : (INSTRUCTION)mov (OPERAND_REG)a (OPERAND_REG)b
Lexer Item 36 : (INSTRUCTION)add (OPERAND_IMM)#1
Lexer Item 37 : (INSTRUCTION)mov (OPERAND_REG)b (OPERAND_REG)a
Lexer Item 38 : (INSTRUCTION)b (OPERAND_ADDR).iter
Lexer Item 39 : (LABEL).str: (KEYWORD)dpw (NAME)hello_world_str (VALUE)"HELLO
    WORLD"
```

Přeložený soubor (Výstup)

```
-- Program memory written using ASSEMBLER 1.00 : 14:19:37 |
   Thursday, April 5, 2017
constant rom : memory_t := (
```

```
0 => "01" & x"0070", -- cms #0
1 => "11" & x"021C", -- mov b, .str'offset
2 => "00" & x"0C84", -- mov a, #200
3 => "00" & x"8281", -- sto #40
4 => "01" & x"00D3", -- call .print_string
5 => "10" & x"8017", -- b .halt
6 => "10" & x"0281", -- lod #40
7 => "11" & x"FEF1", -- sto #CDR_REG
8 => "11" & x"83E6", -- cndx prog
9 => "01" & x"0015", -- lod b
10 => "01" & x"8291", -- sto #41
11 => "11" & x"8610", -- slr #6
12 => "00" & x"03FA", -- and #0x3F
13 => "01" & x"8053", -- reteq
14 => "11" & x"1C0B", -- or #0x1C0
15 => "10" & x"FEE1", -- sto #CHR_REG
16 => "10" & x"0281", -- lod #40
17 => "00" & x"0018", -- add #1
18 => "11" & x"FEF1", -- sto #CDR_REG
19 => "00" & x"8281", -- sto #40
20 => "11" & x"0291", -- lod #41
21 => "00" & x"03FA", -- and #0x3F
22 => "01" & x"8053", -- reteq
23 => "11" & x"1C0B", -- or #0x1C0
24 => "10" & x"FEE1", -- sto #CHR_REG
25 => "10" & x"0281", -- lod #40
26 => "00" & x"0018", -- add #1
27 => "11" & x"FEF1", -- sto #CDR_REG
28 => "00" & x"8281", -- sto #40
29 => "00" & x"0005", -- mov a, b
30 => "00" & x"0018", -- add #1
31 => "01" & x"000D", -- mov b, a
32 => "00" & x"8187", -- b .iter
33 => "01" & x"0A25", -- dpw hello_world_str, "HELLO WORLD"
34 => "11" & x"0B2C", -- dpw hello_world_str, "HELLO WORLD"
35 => "11" & x"0BC2", -- dpw hello_world_str, "HELLO WORLD"
36 => "11" & x"0DEF", -- dpw hello_world_str, "HELLO WORLD"
37 => "00" & x"0CAC", -- dpw hello_world_str, "HELLO WORLD"
38 => "00" & x"0900", -- dpw hello_world_str, "HELLO WORLD"
others => "00" & x"0000");
```

Tabulka 5.5: Výstupní VHDL include soubor

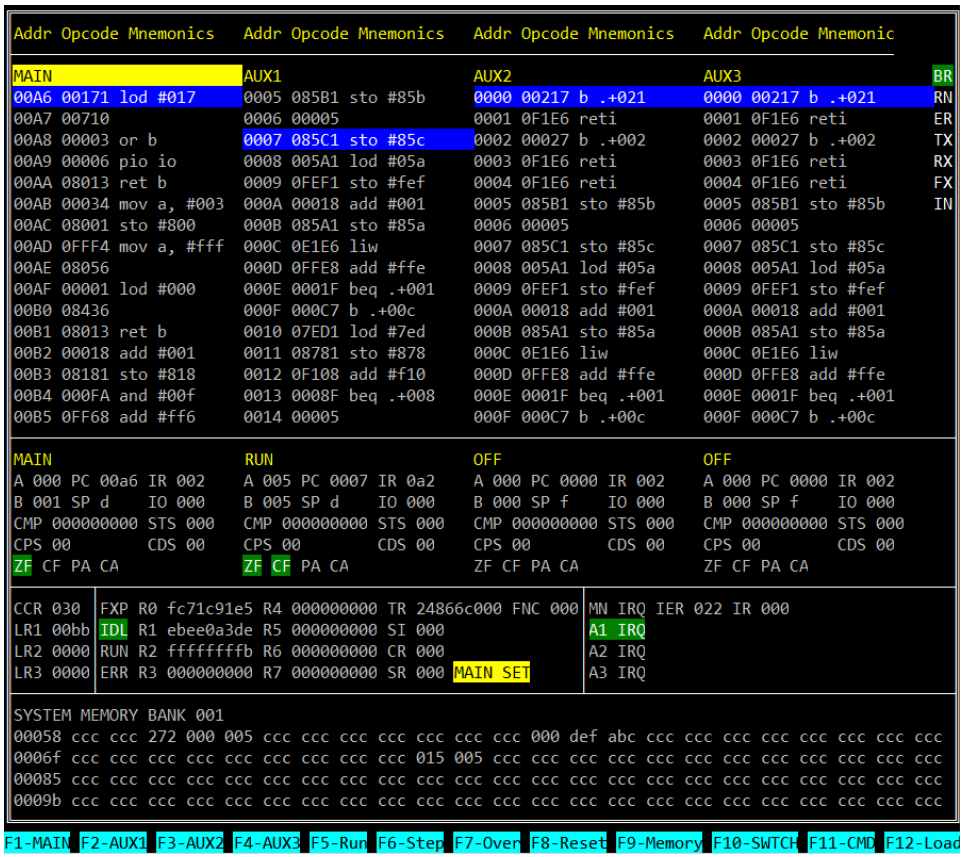
5.2 Emulátor

Pro potřeby vývoje počítače spolu s překladačem jazyka symbolických adres bylo rovněž nutno implementovat emulátor, který by umožnil efektivně testovat software a odhalovat principiální chyby návrhu. Ve skutečnosti je extrémně obtížně napsat bez softwarového emulátoru složitější program cí-

lený na jinou platformu, než na jaké byl původně vytvořen, neboť nároky na časové prostředky jsou neakceptovatelné. Zejména v případě FPGA *prototypingu* se jedná o kritický faktor, protože jakákoliv změna v designu (a tedy i aktualizace softwaru v programové paměti) je následována kompletní syntézou návrhu a routováním. Emulátor nám umožňuje tento nepřípustný krok vynechat a namísto toho zdrojový kód testovat softwarově. Vstupem aplikace emulátoru je konfigurační soubor, který obsahuje všechny informace nutné pro spuštení aplikace v současné verzi. Výrazy rozpoznané emulátorem jsou shrnuty v tabulce níže.

[H] Výraz	Význam
CONFIGURATION STATEMENTS	
magic_breakpoint	Pokud zapnuto, vykonání *NOP-like* instrukce B .+0 uvede emulátor do *breakpoint* stavu.
break_start	Pokud zapnuto, aplikace bude spuštěna v *breakpoint* stavu a vykonávaný program tak bude ihned pozastaven.
assembler_path	Absolutní nebo relativní cesta k překladači jazyka symbolických adres.
assembler_args	Argumenty příkazové řádky překladače a to zejména vstupní soubor. Ovládání překladače je rozebráno výše v sekci (5.1).

Tabulka 5.7: Definice klíčových slov konfiguračního souboru emulátoru
Emulátor přenastaví konzolové okno na nativní rozlišení počítače, tedy 100× 40 [řádek× sloupec]. Pokud ***break_start*** = TRUE pak se emulátor zastaví na první instrukci a zobrazí se ihned okno *monitoru* emulátoru.



Obrázek 5.3: Emulator Monitor

Hlavní dialog monitoru obsahuje převedším okamžitý disassembler pro každou výkonnou jednotku a soupis registrů s aktuálními hodnotami. Zde lze program pohodlně krokovat a rovněž v něm pomocí kurzorových kláves listovat. Níže jsou pak UI komponenty pro koprocesor spolu s CCR a LOAD registry. Po pravé straně vidíme současný stav přerušení a na samém konci dialogu náhled naposledy změněné datové paměti.

Poznámka. Na zachyceném snímku si můžeme povšimnout, že jednotky MAIN a AUX1 jsou zapnuty, zatímco AUX2 a AUX3 suspendovány. Koprocesor je v IDLe stavu a jednotka AUX1 je právě uprostřed vykonávání ISR. Rovněž vidíme, že UI komponenta koprocesoru právě zobrazuje *stínované* kopie registrů pro jednotku MAIN.

Aplikace dále umožňuje přímo psát programy v jednoduchém textovém editoru, který zvýrazňuje (*syntax highlighting*) jednotlivá klíčová slova a rovnou je překládat prostřednictvím implementovaného překladače popsaného v oddíle(5.1).

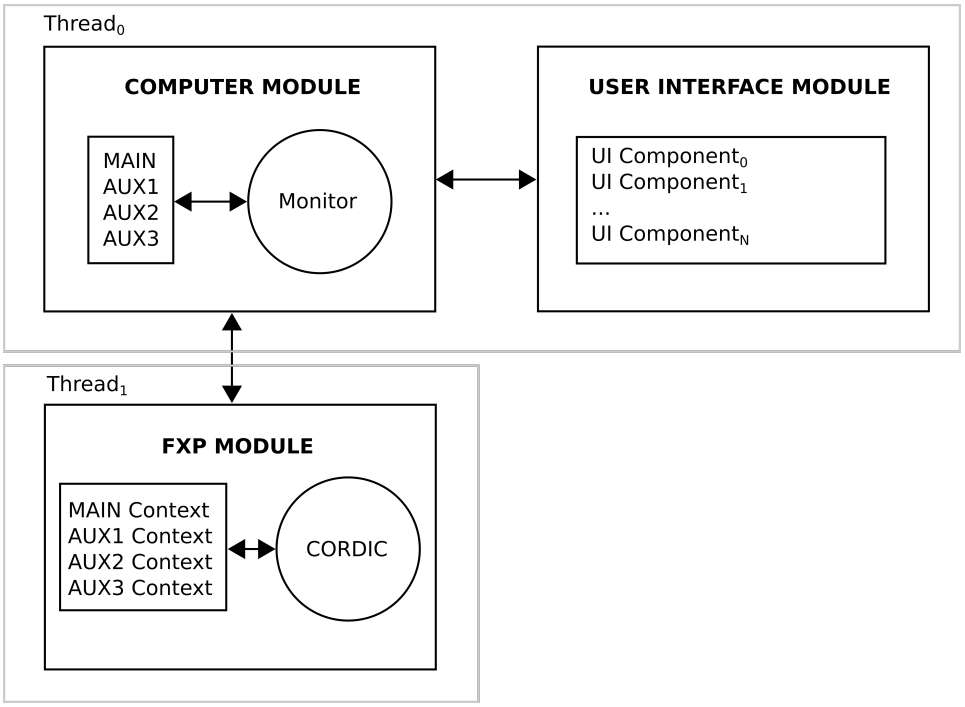
Samotné ovládání programu je velmi intuitivní – z historického hlediska je uživatelské rozhraní totožné se všemi textově orientovanými programy.

5.2.1 Struktura

Emulátor je sestaven z modulů, které zhruba odpovídají jejich hardwarové implementaci. Program je *více vláknový*, čímž je zachována konkuretnost výpočtů výkonných jednotek a koprocesoru. Integrálním prvkem emulátoru je tzv. *Monitor* – komponenta, kolem které jsou postaveny veškeré moduly (5.4) aplikace. *Monitor* je odpovědný za následující úkony:

- *Krokování* (single-stepping) či volný běh programu.
- Aktualizace emulovaného video výstupu.
- Aktualizace a překreslování individuálních komponent uživatelského rozhraní a to v případě, že zobrazená hodnota se liší od hodnot současných.
- Směrování virtuálních kláves standardního vstupu (STDIN) ovladači emulovaného počítače či vlastní aplikaci.

Níže je vyobrazeno zjednodušené schéma *propojení* funkčních modulů.



Obrázek 5.4: Struktura emulátoru

Níže je uveden velmi stručný seznam komponent aplikace.

Procesor

Dle návrhu hlavní procesor sestává ze čtyř výkonných jednotek. Softwarovou implementaci lze nalézt v hlavním souboru *Computer*. Emulátor nejprve sestaví dekódovací *lookup* tabulku pro každou z instrukcí. Instrukce zdrojového programou jsou interpretovány. Komponenta je rovněž odpovědná za arbitraci datové paměti, obsluhu přerušení a komunikaci s matematickým koprocesorem FXP.

Hlavní smyčka *vlákna* hlavního procesoru sestává z následující sekvence událostí:

1. Instrukční slovo i-té jednotky je vyčteno a dekódováno.

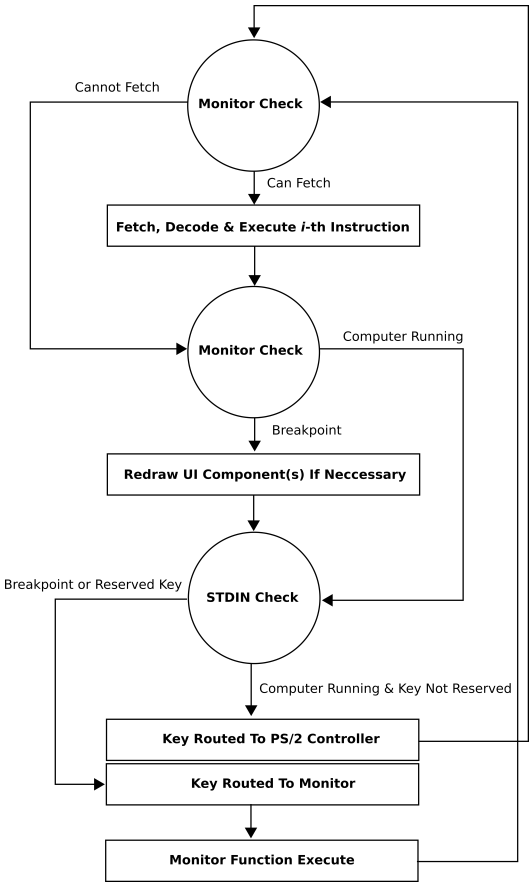
2. Instrukční slovo je vykonáno a interní stav jednotky je aktualizován.

3. Pokud je program suspendován, počítač je v *breakpoint* režimu a podmínka obnovení uživatelského rozhraní může být spočtena. V případě potřeby je pak daný prvek (či prvky) uživatelského rozhranní překreslen.

4. Vstup klávesnice je přečten a předán ovladači PS/2 nebo vlastní aplikaci, a to v závislosti na hodnotě[2] vyčtené klávesy.

Schematicky je tento proces znázorněn níže

[2]Stisk některých kontrolních kláves je PS/2 ovladačem počítače ignorován a přímo předán aplikaci emulátoru. Seznam *rezervovaných* kláves je k nalezení v závěru oddílu.



Obrázek 5.5: Emulátor hlavního procesoru

Datová paměť

Datová paměť je sestavena z 2048 12-bitových slov a v souladu s návrhem implementuje MMIO (*Memory Mapped Input and Output*) a další paměťově mapované registry. Datová paměť je jednou z dependencí modulu *Computer*.

Koprocesor

Koprocesor softwarově implementuje univerzální CORDIC rovnice popsané v oddílu (3.6).

Cordic

Modul obsahuje jádro CORDIC počítače. Princip a implementace CORDIC mechanismu již byly popsány v kapitole (3.6) a proto se jimi již zde zabývat nebudeme. Jedna z dependencí modulu *Fxp*.

5.2.2 Systematické chyby

Softwarová emulace je ze zřejmých důvodů uplně rozdílná od skutečné hardwarové implementace. Klíčové hardwarové aspekty jako je automatická totální konkuretnost všech procesů a přísné elektronické restrikce na počty signálů, délku kritické cesty či časování obecně, jsou v případě softwarové emulace zcela ignorovány. Běžně používané softwarové konstrukty jsou často v hardwarovém provedení nepoužitelné z důvodu neakceptovatelné utilizace zdrojů či neschopnosti vyhovět časovacím restrikcím návrhu. Už pouze jednoduchý iterativní variabilní bitový posun o i pozic, který lze v softwarovém kódu zapsat například takto:

```
unsigned int var = 1;
for(unsigned int i = 0; i < 8; i++)
{
        var <<= i;
}
```

je z hlediska hardwarové implementace často nepřípustný, neboť popisuje tzv. *barrel shifter*, rotační posunovač. Uvedený příklad, byť triviální, nabývá extrémní důležitosti, neboť například CORDIC jednotka matematického koprocesoru implementuje tento typ bitového posunu při každém kroku svého iterativního procesu. Vstupní slovo o délce n bitů je připojeno na vstupy n multiplexorů, každý šířky n–1[7]. Čistě teoreticky je počet multiplexorů potřebných k realizaci rotačního posunovače roven $n \log_2 n$[7], nicméně skutečné číslo záleží na dané technologii. Multiplexory jsou na čipu organizovány do logických celků a jsou fixní délky. Synthetizační nástroj tak musí najít optimální konfiguraci vzájemného kaskádního propojení dostupných multiplexorů, aby vyhověl návrhu. V případě konkrétní architektury chipu řady Spartan6 použité při vývoji se toto číslo pro šířku 33-bitů vyšplhalo na přibližně 750 multiplexorů. Návrh proto musel být zcela přeorganizován, aby mohly být bity posunuty vždy o *konstatní* hodnotu.

Softwarová emulace není však těmito principy vázána a behaviorálně se oba modely mohou chovat stále velmi *podobně*. Systematické chyby v emulaci tedy jsou:

1. Instrukce jsou jednotkami vykonávány sekvenčně, namísto konkurentě. Toto je patrné již ze schématu (5.5). Nejprve je vykonána operace na jednotce MAIN, AUX1, AUX2 a konečně AUX3. Vzhledem k tomu, že hostitelský počítač je schopen vykonávat sekvenci těchto operací velmi rychle, je rozdíl nepostřehnutelný. Rovněž sychronizace dat mezi výkonnými jednotkami je v hardwarové implementaci atomická a přístup do datové a programové paměti je sekvenčně arbitrován. Až na určitou (byť z praktického hlediska zcela zanedbatelnou) prodlevu jsou oba modely behaviorálně zcela totožné.

2. Automatická arbitrace dat vycházející z přirozené atomicity softwarových výrazů.
3. Absence systémové peripherální sběrnice.
4. Emulátor nevykonává program v *reálném čase*. Program, který počítá čas například dle předkpokládaného kmitočtu hlavních systémových hodin, selže.
5. Periférie je často veliký problém správně emulovat, zvláště pak jejich vstupy.

Poznámka. UART není v současné verzi emulátoru podporován.

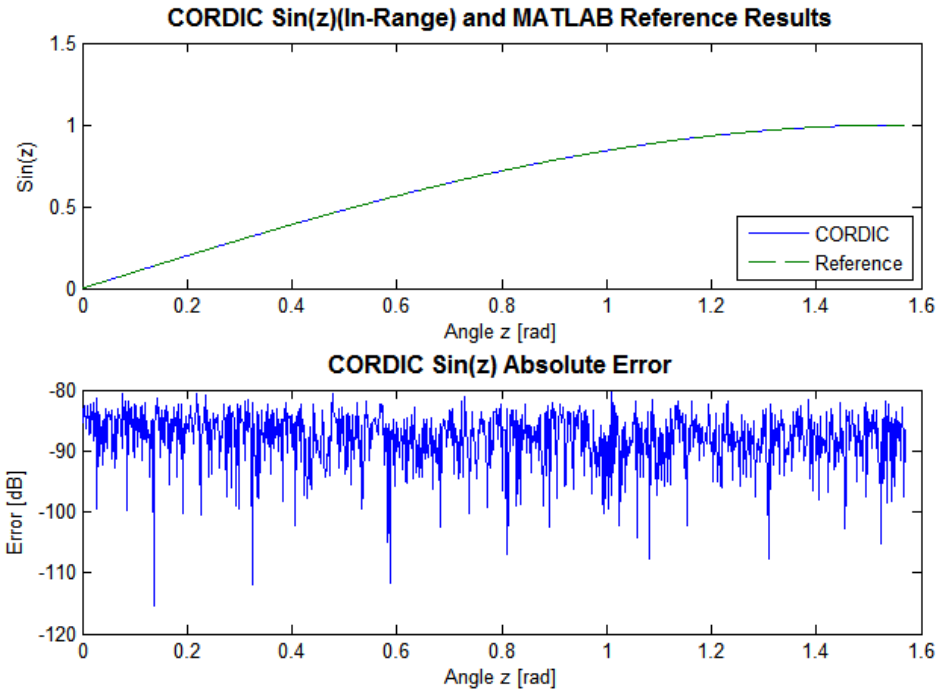
Kapitola 6

Simulace a ověření návrhu

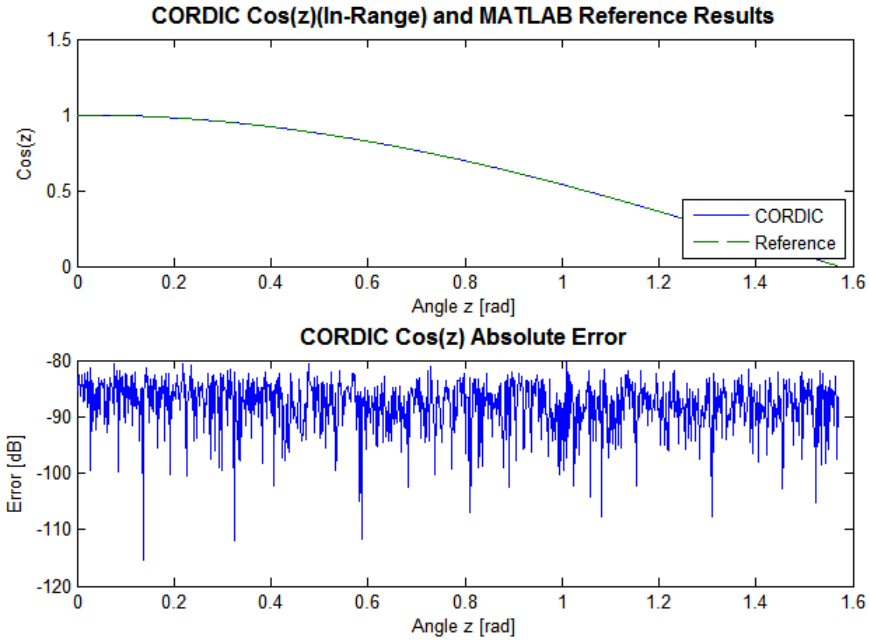
V této kapitole se zaměříme především na ověření CORDIC jednotky, neboť je fundamentálním výpočetním mechanismem celého koprocesoru. Předmětem druhého oddílu kapitoly je pak vzorový program počítače, jehož plný zdrojový lze vzhledem k jeho délce nalézt v příloze (B).

6.1 Koprocesor

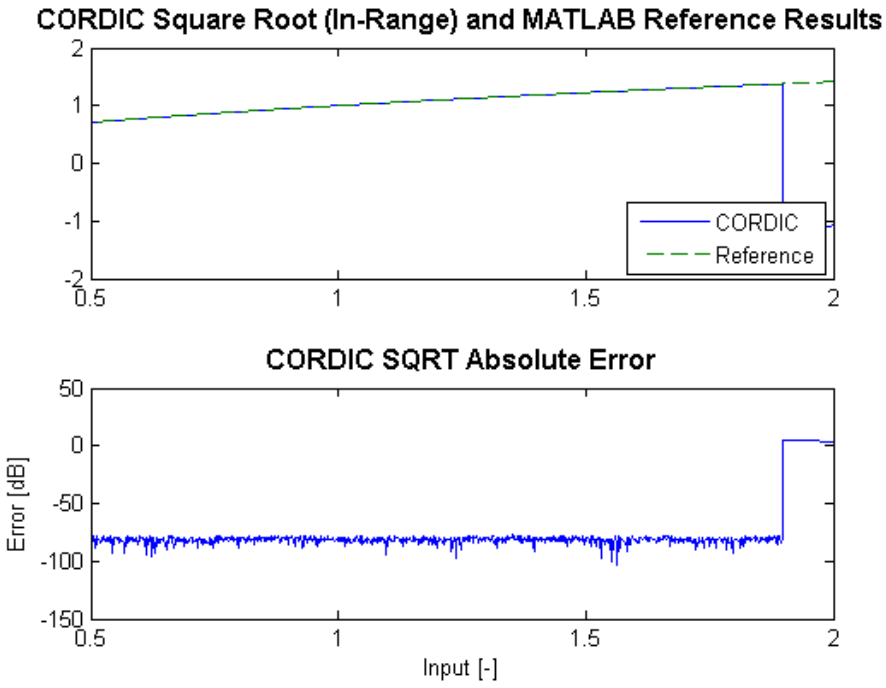
Přesnost výpočtů CORDIC funkcí matematického koprocesoru byly ověřeny prostřednictvím MATLABu. Celkem bylo porovnáno vždy 1000 ekvidistantně vzdálených vstupů funkce a absolutní chyba zanesena do grafu.



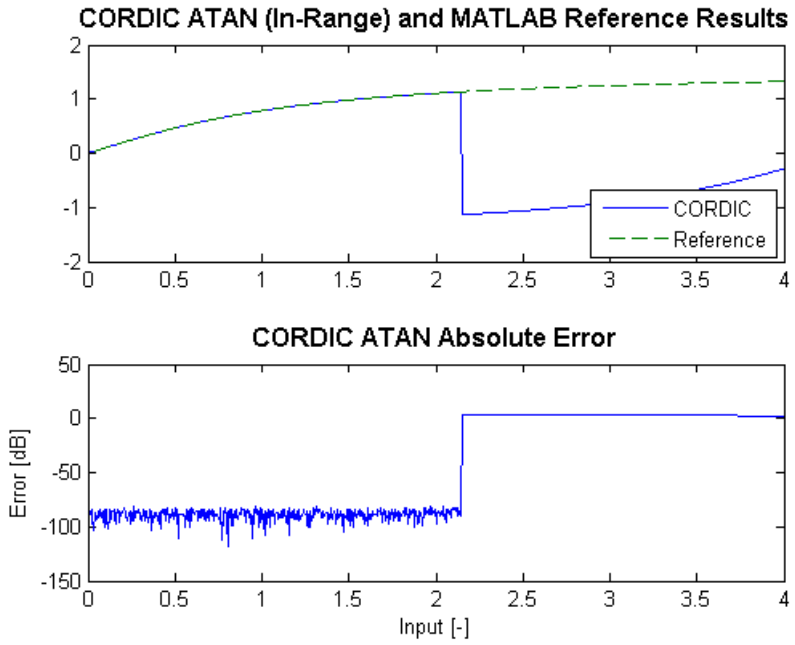
Obrázek 6.1: Ověření CORDIC funkce $\sin(z)$



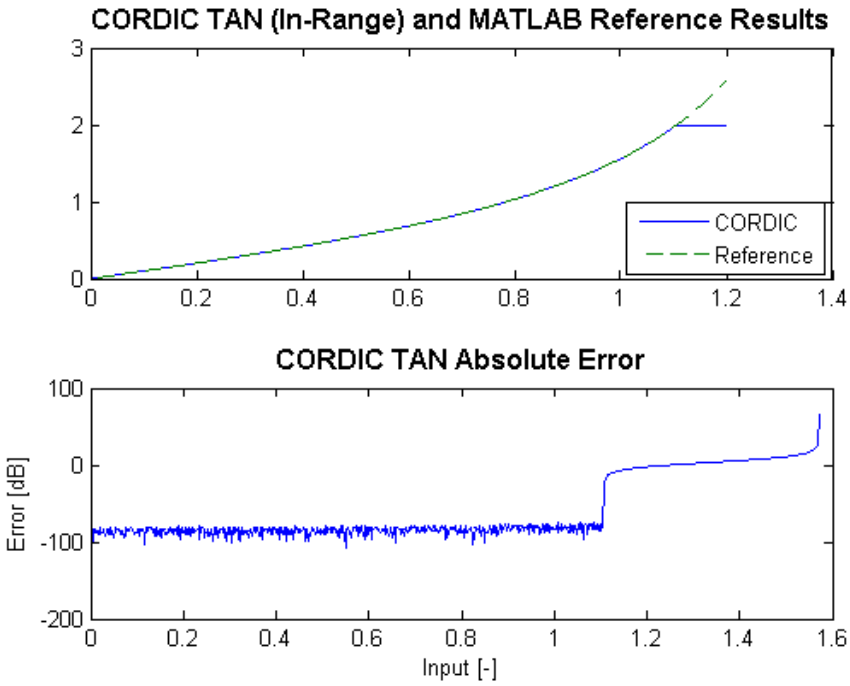
Obrázek 6.2: Ověření CORDIC funkce $\cos(z)$



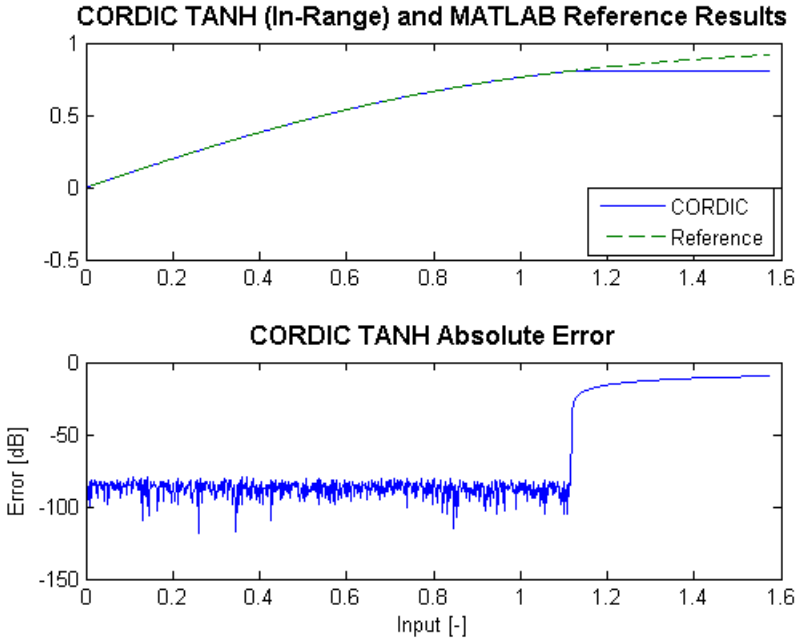
Obrázek 6.3: Ověření CORDIC funkce odmocniny



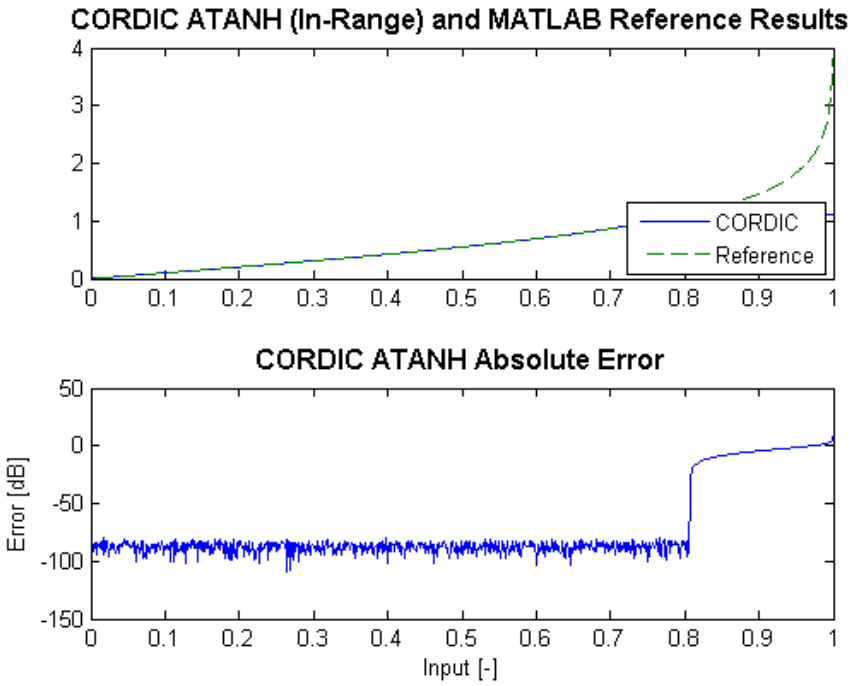
Obrázek 6.4: Ověření CORDIC funkce atan(y)



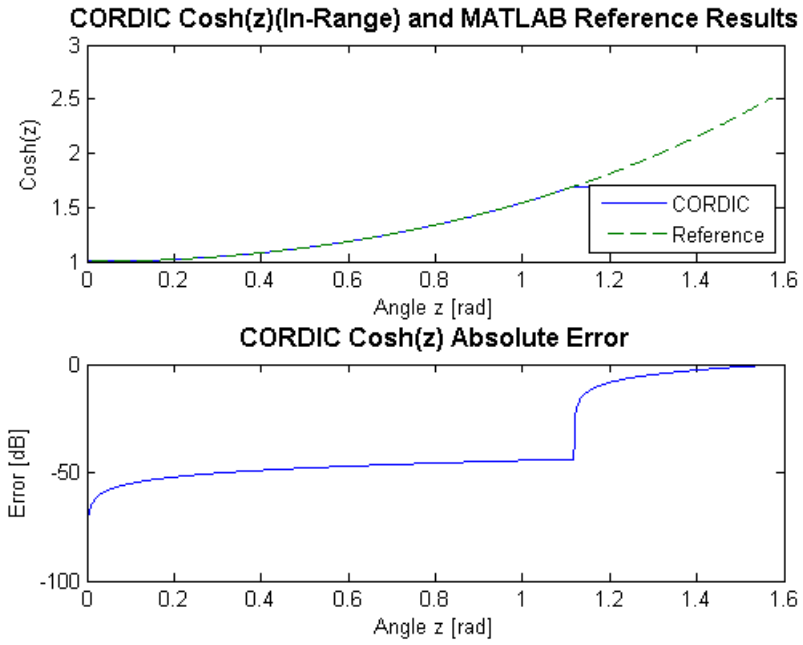
Obrázek 6.5: Ověření CORDIC funkce tan(w)



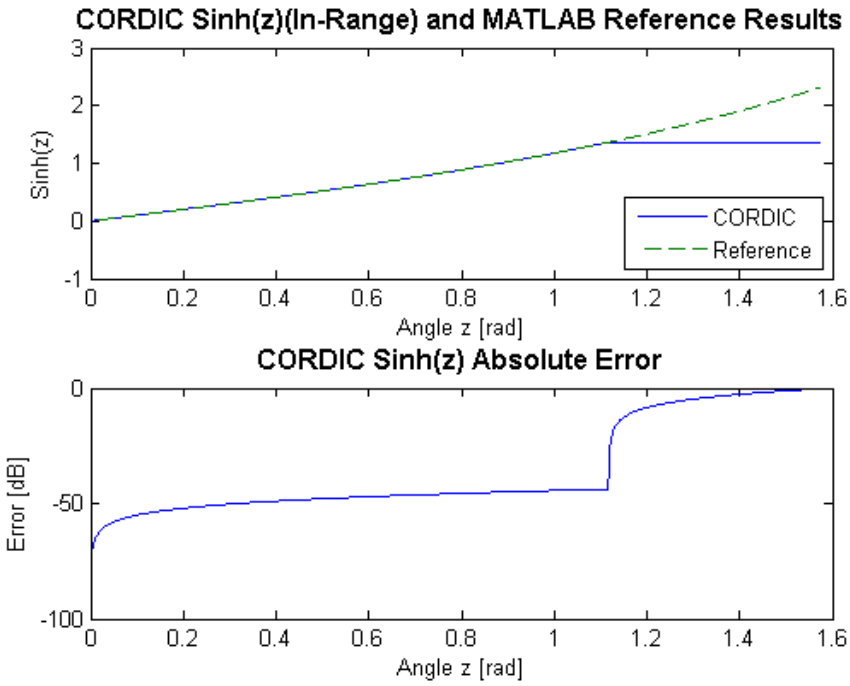
Obrázek 6.6: Ověření CORDIC funkce tanh(w)



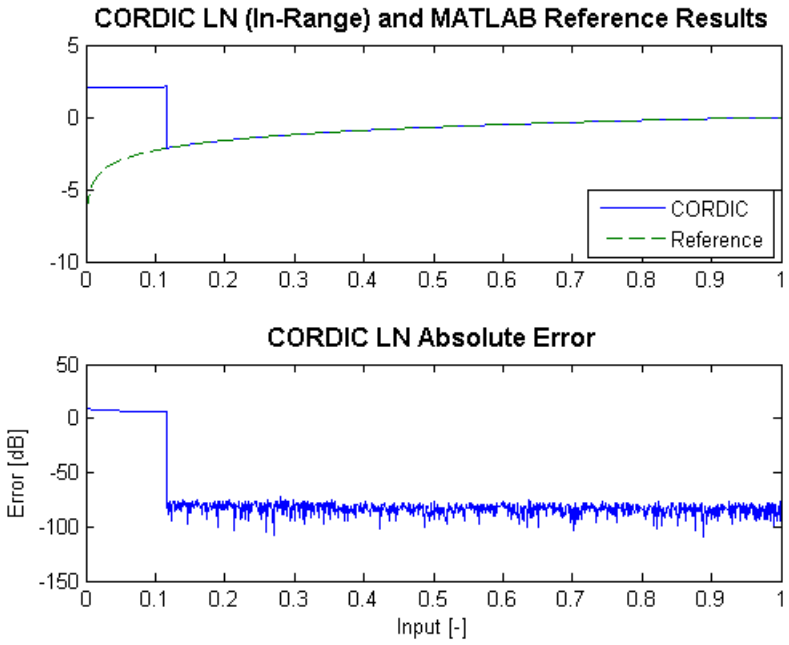
Obrázek 6.7: Ověření CORDIC funkce atanh(z)



Obrázek 6.8: Ověření CORDIC funkce cosh(z)



Obrázek 6.9: Ověření CORDIC funkce $\sinh(z)$



Obrázek 6.10: Ověření CORDIC funkce $\ln(w)$

Z přiložené simulace je patrné, že FXP koprocesor vykazuje v porovnání s referenčními hodnotami MATLABu minimální chybu, která je navíc uniformně rozprostřena přes celou vstupní doménu funkce. Hyperbolické funkce $\sinh(z), \cosh(z)$ vykazují v porovnání s ostatními výpočty největší chybu, což je způsobeno především chybou konvergence univerzálního CORDICU pro hyperbolické souřadnice. Rovněž si můžeme povšimnout zřejmých limitů vstupních úhlů hyperbolických funkcí, které byly zmíněny v oddíle (3.6.2), kde konvergence algoritmu selže úplně. Problémy konvergence a rozšíření vstupní domény úhlů je možné řešit modifikovaným[8] CORDIC schématem speciálně uspůsobeným pro výpočet hyperbolických funkcí.

6.2 Vzorový program

S přihlédnutím k časové náročnosti hardworového návrhu a podpůrného softwaru byl pro počítač napsán pouze jediný vzorový program. Výkonná jednotka MAIN vykonává program z původní bakalářské práce[12] zatímco jednotce AUX1 je předána kontrola nad video výstupem a PS/2 rozhraním.

Program vykonávaný hlavní jednotkou periodicky každou sekundu inkrementuje BCD[1] čítač. Hodnota čítače je v časovém formátu HH:MM průběžně zobrazována na sedmi-segmentovém displeji. Displej je překreslován s frekvencí přibližně 4 kHz aby se předešlo *ghosting* efektu. Auxiliarní jednotka vytiskne na obrazovku nejprve všechny existující znaky a také 36-bitové číslo 0xABCDEF00 v FXP formátu a načež vstoupí do nekonečné smyčky, kde vyčká na přerušení klávesnice. Stisk klávesy je následně převeden na tisknutelný znak a vytištěn na obrazovku.

Hlavním účelem vzorového programu tedy bylo:

1. **V realistických podmínkách ověřit funkčnost arbitrace datové a programové paměti.** Utilizace datové paměti je vždy velmi vysoká a neschopnost včas obsloužit výkonnou jednotku MAIN by vyústila v selhání programu. Vzhledem k vysoké obnovovací frekvenci displeje by tento fakt byl okamžitě zřetelný.

2. **Ověřit funkčnost VGA videa a přerušení.** Program jednotky AUX1 je schopen vytisknout libvolné FXP číslo v datové paměti. Toto je výpočetně velmi náročný proces, protože je neprve nutné číslo převést do BCD notace, aby mohlo být zobrazeno v desítkové soustavě. Stisk klávesy, převedení *scancodu* do tiskunutelného znaku a konečně jeho tisk na obrazovce musí být pro správnou funkčnost plynulý. Opět, vzhledem k bodu 1, jakákoliv chyba ve sdílení zdrojů víceprocesoro-

[1] BCD, z angl. *Binary Coded Decimal*.

vého systému by způsobila viditelné zpomalení tohoto procesu, pokuď ne uplné selhání programu.

Poznámka. Zdrojový kód vzorového programu je možné nalézt v příloze (B).

Kapitola 7

Závěr

Cílem diplomové práce bylo navrhnout víceprocesorový počítač s dedikovaným matematickým koprocesorem. Počítač je schopen nativních trigonometrických a hyperbolických výpočtů na omezeném vstupním intervalu. Poloměr konvergence CORDICu v kruhových souřadnicích lze jednoduše rozšířit užitím příslušných matematických identit. Pro rozšíření poloměru konvergence CORDICu v hyperbolických souřadnicích by však byl nutný individuální přístup ke každé ze zamýšlených funkcí[8]. Dle výsledků automatizovaných testů je přesnost implementovaného matematického koprocesoru vzhledem ke stanoveným požadavkům dostatečná.

Každá z vykonných jednotek počítače, MAIN, AUX1, AUX2 a AUX3, je zpětně kompatibilní s návrhem, který byl náplní bakalářské práce[12].

7.1 Možnosti rozšíření

V následujícím oddílu budou ještě v krátkosti diskutována možná rozšíření návrhu.

7.2 Překladač

Překladač jazyka symbolických adres lze v současné implementaci velmi snadno rozšířit o řadu vysokoúrovňových maker. Všechny následující výrazy jsou *lexerem* snadno rozpoznatelné a tříditelné do tokenů. Zbývalo by tedy pouze přizpůsobit fázi *parsování*, jejímž výstupem je již skutečný strojový kód. Obecně platí:

- Makro může obsahovat libovolnou instruknci či další makro. Všechny makro výrazy lze plynule mísit se strojovými instrukcemi, podobně jako tomu bylo v ranných verzích programovacího jazyka BASIC[10].

- Každé makro sestává z jednoho nebo více rezervovaných klíčových slov definovaných dále v textu.
- Dle gramatických pravidel jazyka je každému makro výrazu vyhrazen právě jediný řádek ve zdrojovém kódu[1].

Poznámka. Připomeňme ještě ve stručnosti dosud implementovaná makra a direktivy, které lze nalézt v oddilu5.1.1.

7.2.1 Datové typy

Gramatika implemetuje dva datové typy:

Typ	**Rozsah**	**Poznámka**
INTEGRAL	0–4095	Celočíselný typ
STRING	Neomezen	Řetězec

Datový typ je vždy implicitní a je automaticky odvozen v závislosti na zdrojovém operandu.

Poznámka. Definice proměnné typu STRING je velmi naročná na utilizaci datové paměti, neboť každý ze znaků musí být individuálně zapsán. Definice proměnné

```
VARIABLE str = "HELLO"
```

vygeneruje následující sekvenci instrukcí:

```
mov     a, #"H"
sto     #str
mov     a, #"E"
sto     #str + 1
mov     a, #"L"
sto     #str + 2
sto     #str + 3
mov     a, #"O"
sto     #str + 4
mov     a, #0
sto     #str + 5
```

Paměťový alokátor překladače rovněž musí pro nově vzniklý řetězec rezerovovat dostatečně velký paměťový blok. Pro konstatní řetězce je tedy doporučeno užití DPW direktivy, která umístí řetězec do programové paměti v době překladu vstupního souboru.

[1] Makra jsou oddělena CRLF *Carriage Return, Line Feed* symbolem.

7.2.2 Proměnné

Syntaxe definice proměnné je totožná s definicí konstanty, liší se pouze klíčové slovo.

```
VARIABLE idtify = value
VARIABLE idtify
```

Na rozdíl od konstanty CONSTANT lze proměnnou VARIABLE nejprve *deklarovat* a hodnotu přiřadit později. Hodnota konstanty musí být přiřazena právě v době deklarace (konstanta je deklarována a definována současně). Definice jsou zpracovány v prvním průchodu (*pass*) zdrojového kódu a proto je pořadí konstanty CONSTANT irelevantní, na rozdíl od proměnné VARIABLE, která musí být neprve definována, než může být referencována. Nedefinované proměnné budou prekladačem odstraněny. CONSTANT a VARIABLE mají vždy globální kontext (*global scope*) a jsou tudíž viditelné v rámci celé překladové jednotky (*translation unit*).
Přiřazení do proměnné vygeneruje následující kód

```
lod     #VALUE
sto     #VAR
```

VALUE může být libovolný konstatní výraz sestávající z aritmetických a logických operátorů nebo proměnná IDTIFY.

Ternární operátor

```
VARIABLE idtify = (condition) ? [TRUE st.] : [FALSE st.]
```

Poznámka. Zde je prostor pro výrazy [TRUE statement], [FALSE statement] příliš rozsáhlý pro statické generování kódu. Místo toho musí být strojové instrukce automaticky odvozeny v závislosti na zdrojových operandech přiřazení. Ukažme tedy alespoň příklad kódu vygenerovaného pro konkrétní makro příkaz

```
VAR1 = (VAR2 = CONST) ? #0xABC : [#0xDEF];
```

```
; Assuming variables and constants VAR1, VAR2, CONST1
IF VAR2 = CONST1 THEN .true
.false: VAR2 = [#0xDEF]
        b       .done
.true:  mov     a, #0xABC
        sto     VAR1
.done:
```

Ukazatele

Ukazetele proměnných mají následující předpis

```
; Store value to memory location pointed to by idtify
[idtify] = value
; Load value from memory location pointed to by idtify
value = [idtify]
```

Strojový kód vygenerovaný prvním výrazem je

```
lod     #IDTIFY
mov     b, a
lod     #VALUE
sto     b
```

Strojový kód vygenerovaný druhým výrazem je

```
lod     #IDTIFY
mov     b, a
lod     b
```

VALUE může být buď konstanta CONSTANT nebo proměnná VARIABLE.

Podmíněné větvení

Automatizace podmíněného větvení velmi zpřehlední a zjednoduší zdrojový kód. Makro podmíněného větvení je následujícího předpisu:

```
if [condition] THEN label
```

kde *condition* $\in$ {TRUE, FALSE} může nabývat následujích hodnot

```
1. condition = ([variable] [relation] [variable])
2. condition = ([variable] [relation] [constant])
```

- CONSTANT může být libovolný konstatní výraz
- RELATION je z $\{=, >, <, \geq, \leq\}$

Poznámka. Vzhledém k větší délce lze vygenerováné strojové instrukce makra podmíněného větvení najít v příloze (A).

Podmínka *condition* je spočtena zleva doprava pomoci tzv. *líné evaluace* (lazy evaluation). Překladč v rámci optimializace zcela odstraní kód makra pro předem nesplnitelné podmínky (TRUE = FALSE), podobně jako tautologie (TRUE = TRUE), které nahradí celé makro pouze instrukcí nepodmíněného větvení.

For cyklus

Makro může rovněž zahrnovat *for* cyklus.

```
FOR [variable] = [expr] TO [expr] STEP [expr]
[statements]
NEXT [variable]
```

Poznámka. Podobně jako v případě ternárního operátoru bude výstup odvozen v závislosti na vstupních příkazech makro příkazu. Uveďme proto alepoň příklad pro nejčastěji používaný tvar *for* cyklu:

```
FOR VAR1 = VAR2 TO VAR3 STEP VAR3, NEXT VAR1
```

```
; Assuming variables VAR1, VAR2, VAR3
.loop: IF VAR2 = VAR3 THEN .done
       [statements]
       lod      #VAR1
       mov      b, a
       lod      #VAR3
       add      b
       b        .loop
       .done:
```

Funkce a makro příkazy

Překladač může podporovat dedikovaný makro příkaz či přímo funkci. Funkce vygeneruje kód který je ekvivalentní volání procedury pomocí instrukce CALL. Návratová hodnota může být uložena v akumulátoru nebo na dynamicky alokované paměťové pozici.

```
FUNCTION idtify(arguments)
[statements]
return value
END
```

Makro jednoduše zkopíruje veškeré příkazy a instrukce na příslušnou adresu.

```
MACRO idtify(arguments)
[statements]
END
```

Obsah makra je tak replikován při každém použití výrazu.

7.3 Výkonné jednotky

Součástí práce byla implementace překladače jazyka symbolických adres. Dalším krokem by mohla být implementace kompileru vhodného dialektu

jazyka C. Kompiler by mohl být navrhut samostatně nebo by mohlo být využito již existujích gramatických generátorů. Každá z výkonných jednotek počítače z důvodu zpětné kompatibility vychází z původní bakalářské práce. Při zachování stejné architektury by bylo možné navrhnout modifikovanou instrukční sadu, která by usnadnila případný návrh kompileru dialektu jazyka C.

7.4 Výsledky syntézy

Logic Utilization	Used	Available	Utilization
DEVICE UTILIZATION SUMMARY			
Number of Slice Registers	4240	18224	23%
Number of Slice LUTs	8240	9112	90%
Number of LUT-FF pairs	3637	8610	42%
Number of PLL_ADVs	1	2	50%
Constraint	Period Req.	Actual Period	Timing Errors
TIMING SUMMARY			
System Clock	10.000 ns	9.971 ns	0
Pixel Clock	25.000 ns	24.925 ns	0

Tabulka 7.2: Využitá plocha cílového FPGA a podmínky časování

Všechny okrajové podmínky byly splněny a hlavní hodinový signál modifikovaného počítače je stále 100 MHz. Velikost matematického koprocesoru zabírá přibližně polovinu celého návrhu počítače, tj. 4058 LUT tabulek.

Poznámka. Utilizace je platná pro cílový chip Spartan 6 LX16. Systém je pochopitelně možné syntetizovat i na jiných FPGA architekturách – jediným požadavkem je 100 MHz kmitočet hlavního hodinového signálu, *dostatečná* plocha čipu a podpora fázového závěsu PLL.

Příloha A

Makro podmíněného větvení

Níže jsou pro každou z relací uvedeny příklady strojových instrukcí vygenerovaných makrem.

```
; Assuming two variables VAR1 and VAR2
; 1) relation is equality: VAR1 '='VAR2
lod     #VAR1
mov     b, a
lod     #VAR2
not
add     b
add     #1
beq     .label
; 2) relation is greater then or equal:  VAR1 '=>' VAR2
lod     #VAR1
mov     b, a
lod     #VAR2
not
add     #1
add     b
; Carry to Zero Check
mov     b, a
notcy
add     b
add     #1
beq     .label
; 3) relation is less then or equal: VAR1 '=<' VAR2
lod     #VAR1
mov     b, a
lod     #VAR2
not
add     #1
```

```
add     b
; Carry to Zero Check
mov     b, a
notcy
add     b
add     #1
beq     .+1
b       .label
; 4) relation is less then: VAR1 '<' VAR2
lod     #VAR1
mov     b, a
lod     #VAR2
not
add     #1
add     b
; Carry to Zero Check
mov     b, a
notcy
add     b
beq     .label
; 5) relation is grater then: VAR1 '>' VAR2
lod     #VAR1
mov     b, a
lod     #VAR2
not
add     #1
add     b
; Carry to Zero Check
mov     b, a
notcy
add     b
beq     .+1
b       .label
```

Příloha B

Vzorový program

```
ORG 0
;** ****************************************
;** 12 - Bit General Purpose Computer
;** ****************************************
;** Title    : Unit Test
;** Purpose : This is a sample program for the computer.
;** Note     : Target device clock is assumed to be 100 MHz.
;** ****************************************
;** Version : 1.0
;** Date     : 9 January 2018
;************************************************
GENERATE VHDL
;************************************************
;* Program Mainloop
;************************************************
; Compare Register
constant TIM_CMPL = #0x7FB
constant TIM_CMPM = #0x7FA
constant TIM_CMPH = #0x7F9
; Status Register
constant TIM_STS  = #0x7FC
; Computer Control Register and Load Register
constant CCR = #0x7F7
constant LR1_L = #0x7F6
constant LR1_H = #0x7F5
; Video Registers
constant DCR = #0x7F0
constant CDR = #0x7EF
constant CHR = #0x7EE

constant mem_hours = #20;
constant mem_mins  = #21;
constant mem_secs = #25
```

```
    b .ivt_skip
    reti        ; MAIN Interrupt Service Routine Vector
    b .aux1_isr ; AUX1 Interrupt Service Routine Vector
    reti        ; AUX2 Interrupt Service Routine Vector
    reti        ; AUX3 Interrupt Service Routine Vector

.aux1_isr:
    sto     #91 ; store A
    mov     a, b
    sto     #92 ; store B

    lod     #90
    sto     #CDR
    add     #1
    sto     #90

    liw
    add     #0xFFE    ; -2
    beq     .aux1_isr_kbd
   ;mov     a, #0     ; clear everything
   ;siw
    b       .aux1_isr_done
.aux1_isr_kbd:
   lod      #0x7ED    ; read keyboard scancode
   sto      #120      ;scancode
   add      #0xF10    ; -0xF0 break & extra code prefix,
      ignore those
   beq      .aux1_isr_done

   mov      a, b      ; store IER state
   sto      #121
   call     .translate_scancode  ; B = printable character
   ;mov a,  #0x21
   mov      a, b
   or       #0x1C0
   sto      #CHR
   lod      #121      ; restore IER state
   mov      b, a
   ;mov     a, #0
   ;siw               ; clear everything
   .aux1_isr_done:
   mov      a, #0
   siw               ; clear everything

   lod      #92
   mov      b, a
   lod      #91
   reti
```

```
.ivt_skip:
; turn on the auxiliary unit #1
  cms     #0
  lod     #CCR
  or      #8          ; set the IDL bit
  and     #0xFFE      ; clear STA bit
  sto     #CCR
  mov     b, a
  mov     a, .unit1_entry'high
  sto     #LR1_H
  mov     a, .unit1_entry'low
  sto     #LR1_L
  mov     a, b
  or      #1          ; set STA bit
  sto     #CCR
  and     #0xFF6      ; clear IDL and STA bits
  sto     #CCR
; UNIT #1 is ON now
...
; MAIN Unit Runs Its Code Here.
...
;*****************************************************
;*                    BCD Increment                  *
;*  Input: A BCD number                              *
;*  Output: A incremented BCD number                 *
;*****************************************************
constant temp_a = #24;
; add 1, if low nibble is A or greater, add 6
.bcd_increment:
  add     #1
  sto     #temp_a
  and     #0x00F
  add     #0xFF6      ; -10
  lod     #temp_a
  beq     .bcd_carry
  ret
.bcd_carry: add #6
  ret
;*****************************************************
;*                  UNIT #1 ENTRY POINT              *
;*****************************************************
.unit1_entry:
  cms     #0
  mov     a, #600 ; 6th row
  sto     #90     ; keyboard cursor
  mov     b, .lbl'offset
  mov     a, #200
  sto     #40
  call    .print_string
```

```
; print all characters
  mov     a, #0
  sto     #41  ; char to be printed
  mov     a, #300 ; print to 4th row
.print_all:             sto #CDR
  add     #1 ; advance cursor
  mov     b, a
  lod     #41 ; load character
  or      #0x1C0 ; set attribute
  sto     #CHR
  and     #63 ; clear attribute
  add     #1  ; next character
  sto     #41
  add     #0xFC0 ; -64
  beq     .halt ; done
  mov     a, b
  b       .print_all

  mov     a, #400; print to 5th row
  sto     #40
  mov     a, #0xABC
  sto     #53
  mov     b, #53
  call    .print_numhex

  mov     a, #0x000
  sto     #100
  mov     a, #0xDEF
  sto     #101
  mov     a, #0xABC
  sto     #102
  mov     b, #100

  mov     a, #500 ; print to 6th row
  sto     #40
  call    .print_fxpnum

  mov     a, #600
  sto     #90

  mov     a, #0
  mov     b, #5 ; master and kbd irq on
  siw

.halt:    b .halt
```

```
;*****************************************************
;*          Print String From Program Memory         *
;*       Input: B string pointer, [40] cursor        *
;*       Output: None                                *
;*****************************************************
.print_string:
  lod     #40
  sto     #CDR          ; Set cursor
  cndx    prog          ; Read program
.print_string_iter:     lod b                   ; Load lower
    12 bits of program word
  sto     #41           ; Temporarily store the packed word
  slr     #6            ; 1st packed character
  and     #0x3F         ; Zero?
  reteq
  or #    0x1C0
  sto     #CHR          ; Store 1st character in Video
    Memory

  lod     #40
  add     #1
  sto     #CDR          ; Advance cursor
  sto     #40

  lod     #41
  and     #0x3F         ; 2nd packed character
  reteq                 ; Zero?
  or      #0x1C0
  sto     #CHR          ; Store 2nd character in Video
    Memory

  lod     #40
  add     #1
  sto     #CDR           ; Advance cursor
  sto     #40

  mov     a, b
  add     #1              ; Increment pointer
  mov     b, a
  b       .print_string_iter
```

```
;**************************************
;* Print Number From Data Memory In Hex *
;* Input: B pointer, [40] cursor        *
;* Output: None                         *
;**************************************
; Mapping 4 bits => 6 bits
; 0x0 => 0x10, 0x1 => 0x11, 0x2 => 0x12, 0x3 => 0x13
; 0x4 => 0x14, 0x5 => 0x15, 0x6 => 0x16, 0x7 => 0x17
; 0x8 => 0x18, 0x9 => 0x19, 0xA => 0x21, 0xB => 0x22
; 0xC => 0x23, 0xD => 0x24, 0xE => 0x25, 0xF => 0x26
.print_numhex:
  cndx      data
  mov       a, #3
  sto       #51       ; Store nibble count
  lod       #40
  add       #0xFFF    ; -1
  sto       #40       ; Decrement cursor
  lod       b
  sto       #50       ; Store word
  slr       #8        ; High nibble
.print_numhex3:
  mov       b, a
  lod       #40
  add       #1
  sto       #40
  sto       #CDR      ; Advance cursor
  mov       a, b
  add       #0xFF6    ; -10
  sto       #52
  mov       b, #0xFFF
  movcy     a, b      ; if nibble > 10
  add       #1        ; carry to zero check
  beq       .print_numhex_hi
; Printing value < 0xA
.print_numhex_lo:
  add       #0x19     ; -1 + 10 + 0x10, converted value
  b         .print_numhex_skp
.print_numhex_hi:
; Printing value >= 0xA
  lod       #52
  add       #1        ; nibble = 9?
  beq      .print_numhex_lo
  lod       #52
  add       #0x21     ; Converted value
.print_numhex_skp:
  or        #0x1C0    ; Set attribute
  sto       #CHR
```

```
; Printed and advanced pointer
  lod       #51
  add       #0xFF     ; Decrement nibble count
  sto       #51
  add       #0xFFE    ; -2?
  beq       .print_numhex2
  lod       #51
  add       #0xFFF    ; -1?
  beq       .print_numhex1
  ret
.print_numhex2:
; Print mid nibble
  lod      #50
  slr      #4
  and      #0xF
  b        .print_numhex3
.print_numhex1:
; Print low nibble
  lod      #50
  and      #0xF
  b .print_numhex3
;*********************************************
;* Convert Absolute Value of FXP Number to BCD *
;* Input:  B pointer to LSW                    *
;* Output: #63 to #66 BCD Converted Number     *
;* Destroys: #60 to #62, #70 to #79            *
;*********************************************
; Number of bits needed to store the result: n + 4*ceil(
   n/3)|n=32 = 76 bits
; This amounts to 7 words 12-bit words.
; Algorithm: Double dabble / shift-and-add-3
; 1. Reserve scratch pad of 76 bits.
; 2. Before shifting add 3 to any BCD digit that is
    greater then 4
; 3. On each iteration left shift scratch pad by 1 bit.
; 4. Iterate n-times, where n is the width of the source
    number (32 here).
; SCRATCH PAD
   INPUT
; 10^9 10^8 10^7 10^6 10^5 10^4 10^3 10^2 10^1 10^0 MSB
                                          LSB
; 0000|0000 0000 0000|0000 0000 0000|0000 0000
   0000|000000000000 000000000000 000000000000
; WORD6     WORD 5          WORD 4          WORD 3
   WORD 2        WORD 1        WORD 0
; (4 bits)
.con_bcd:
```

```
;************************************
;* Initialize scratch pad & variables *
;************************************
; Copy the input to #60 to #62
  cndx      data
  lod       b
  sto       #60
  mov       a, b
  add       #1
  mov       b, a
  lod       b
  sto       #61
  mov       a, b
  add       #1
  mov       b, a
  lod       b
  sto       #62
; Now zeroize scratch pad words 3 to 6 (#63 to #69)
  mov       a, #0
  sto       #63
  sto       #64
  sto       #65
  sto       #66
; Begin the iteration
  mov       a, #36
  sto       #70         ; iteration variable (n=32)
  mov       a, #2
  sto       #71         ; iteration variable, word count
  mov       a, #63
  sto       #72         ; output scratch pad pointer
;*******************************
;* Check if any BCD digit > 4  *
;*******************************
.con_bcd_check:
  lod       #72
  mov       b, a
  lod       b           ; load word (three digits)
  sto       #73         ; temporarily store the word

  and       #0xF        ; low digit
  sto       #74
  add       #0xFFB      ; -5
  sto       #81         ; nibble = 4?
  mov       b, #0xFFF
  movcy     a, b        ; if low nibble > 4
  add       #1          ; carry to zero check
  mov       a, #0       ; 0th digit
```

```
  sto     #75
  ;beq    .con_bcd_inc
  call    .con_bcd_inc
  lod     #73
  slr     #4
  and     #0xF        ; mid digit
  sto     #74
  add     #0xFFB      ; -5
  sto     #81         ; nibble = 4?
  mov     b, #0xFFF
  movcy   a, b        ; if mid nibble > 4
  add     #1          ; carry to zero check
  mov     a, #1       ; 1st digit
  sto     #75
  ;beq    .con_bcd_inc
  call    .con_bcd_inc
  lod     #73
  slr     #8
  and     #0xF        ; high digit
  sto     #74
  add     #0xFFB      ; -5
  sto     #81         ; nibble = 4?
  mov     b, #0xFFF
  movcy   a, b        ; if high nibble > 4
  add     #1          ; carry to zero check
  mov     a, #2       ; 2nd digit
  sto     #75
  ;beq    .con_bcd_inc
  call    .con_bcd_inc
;************************************
;* Advance the output scratch pointer *
;************************************
.com_bcd_next:
  lod     #72
  add     #1
  sto     #72
  add     #0xFBD      ; -67?
  beq     .con_bcd_shift
  b       .con_bcd_check
;*********
;* Add 3 *
;*********
.con_bcd_inc:
  beq     .con_bcd_inc_go
  ret
.con_bcd_inc_go:
  lod     #81
```

```
  add     #1          ; nibble = 4?
  reteq
  lod     #74
  add     #3
  sto     #74         ; digit + 3
  lod     #75         ; overwrite the corresponding nibble
  add     #0          ; -0
  beq     .con_bcd_st0 ; low nibble
  add     #0xFFF      ; -1
  beq     .con_bcd_st1 ; mid nible
; high nibble
  mov     a, #0x0FF ; nibble mask
  sto     #80
  lod     #74
  sll     #8
.con_bcd_ovw:
  sto     #74
  lod     #72         ; pointer-to-pointer
  mov     b, a
  lod     b           ; got the current word

  mov     b, a
  lod     #80         ; got nibble mask
  and     b

  mov     b, a
  lod     #74
  or      b
  mov     b, a        ; B = value to write
  lod     #72         ; A = pointer to pointer
  xchg
  sto     b           ; written incremented value
  ret                 ; proceed with remaining nibble
.con_bcd_st0:
  mov     a, #0xFF0 ; nibble mask
  sto     #80
  lod     #74
  b       .con_bcd_ovw
.con_bcd_st1:
  mov     a, #0xF0F ; nibble mask
  sto     #80
  lod     #74
  sll     #4
  b       .con_bcd_ovw
;**********************************
;* Now shift everything to the left *
;**********************************
```

```
.con_bcd_shift:
  mov     a, #0
  sto     #75         ; clear CF
  mov     a, #7
  sto     #79         ; word counter
  mov     b, #59
  mov     a, #78      ; scratch pad pointer-to-pointer
  xchg
  sto     b
  xchg

.con_bcd_shift_iter:
  mov     a, b        ; B = word pointer
  add     #1          ; increment pointer
  sto     #78         ; store pointer
  mov     b, a

  lod     b           ; load word
  sll     #1
  sto     b           ; store shifted word

  lod     #75         ; load old CF
  sto     #76

  mov     b, #75
  mov     a, #0
  sto     b
  mov     a, #1
  stocy   b           ; [75] = current CF

  lod     #78
  mov     b, a
  lod     b           ; got shifted word back
  mov     b, a
  lod     #76         ; got old CF back
  or      b           ; carry in
  mov     b, a
  lod     #78
  xchg
  sto     b           ; result stored
;Decrement word count
  lod     #79
  add     #0xFFF      ; -1
  sto     #79
  beq     .con_bcd_shift_done
  lod     #78
  mov     b, a        ; load word pointer
```

```
  b       .con_bcd_shift_iter
.con_bcd_shift_done:
;***************************
;* Check iteration variable *
;***************************
  lod     #70
  add     #0xFFF     ; decrement
  reteq              ; conversion complete
  sto     #70
; Reset the variables
  mov     a, #2
  sto     #71        ; iteration variable, word count
  mov     a, #63
  sto     #72        ; output scratch pad pointer
  b       .con_bcd_check
;****************************************
;*            Print FXP Number          *
;* Input: B pointer to MSW, [40] cursor *
;* Ouput: None                          *
;* Destroys: #60 to #66, #70 to #81,    *
;*            #200 to #206              *
;****************************************
.print_fxpnum:
;*********************************
;* Copy the input to #200 to #202 *
;*********************************
  cndx    data
  lod     b
  sto     #200
  mov     a, b
  add     #1
  mov     b, a
  lod     b
  sto     #201
  mov     a, b
  add     #1
  mov     b, a
  lod     b
  sto     #202
;************************
;* Print & mask the sign *
;************************
  lod     #40
  sto     #CDR       ; set cursor
  lod     #202
  and     #0x100
  beq     .print_fxpnum_pos
```

```
; print negative sign
  mov     a, #0x1CD
  sto     #CHR
  b       .print_fxpnum_bcd
.print_fxpnum_pos:
; print positive sign
  mov     a, #0x1CB
  sto     #CHR
.print_fxpnum_bcd:
  lod     #40
  add     #1
  sto     #40
  lod     #202
  slr     #9
  and     #1
  sto     #206      ; store the radix bit
  lod     #202
  and     #0x0FF
  sto     #202      ; mask the sign and radix bits
;******************
;* Convert to BCD *
;******************
  mov     b, #200
  call    .con_bcd
;********************
;* Print all digits *
;********************
  lod     #40
  sto     #203      ; save cursor
  mov     a, #67
  sto     #204      ; digit pointer
  mov     a, #4     ; nibble counter
  sto     #205
.print_fxpnum_iter:
  lod     #204
  add     #0xFFF    ; decrement pointer
  sto     #204
  mov     b, a
  call    .print_numhex ; print word
  lod     #203
  add     #3
  sto     #203      ; increment cursor
  sto     #40
; Check nibble counter
  lod     #205
  add     #0xFFF    ; decrement
  reteq             ; done printing
```

```
  sto     #205
;************************
;* Check for radix point *
;************************
  lod     #206
  add     #0
  beq     .print_fxpnum_dec0 ; scale 1e-9
  add     #0xFFF
  beq     .print_fxpnum_dec1 ; scale 1e-6
; Decimal point has been already printed.
  b       .print_fxpnum_iter
; Decimal point after 1st MSW
.print_fxpnum_dec0:
  lod     #205
  mov     b, #0xFFD  ; -3
.print_fxpnum_dec0_1:
  add     b
  beq     .print_fxpnum_dec0_2
  b       .print_fxpnum_iter
.print_fxpnum_dec0_2:
  lod     #203
  sto     #CDR
  sto     #40
  mov     a, #0x1CE
  sto     #CHR
  lod     #203
  add     #1          ; increment cursor
  sto     #CDR
  sto     #203
  sto     #40
  mov     a, #2
  sto     #206        ; decimal written flag
  b       .print_fxpnum_iter
; Decimal point after 2nd MSW
.print_fxpnum_dec1:
  lod     #205
  mov     b, #0xFFE      ; -2
  b       .print_fxpnum_dec0_1
```

```
;************************
; Translate Scan Code Set
; to printable character
; Input: [scancode]
; Output:B printable char
;************************
constant scancode = #120

; Scancodes
constant KEY_A  = #0x21
constant KEY_B  = #0x22
constant KEY_C  = #0x23
constant KEY_D  = #0x24
constant KEY_E  = #0x25
constant KEY_F  = #0x26
constant KEY_G  = #0x27
constant KEY_H  = #0x28
constant KEY_I  = #0x29
constant KEY_J  = #0x2A
constant KEY_K  = #0x2B
constant KEY_L  = #0x2C
constant KEY_M  = #0x2D
constant KEY_N  = #0x2E
constant KEY_O  = #0x2F
constant KEY_P  = #0x30
constant KEY_Q  = #0x31
constant KEY_R  = #0x32
constant KEY_S  = #0x33
constant KEY_T  = #0x34
constant KEY_U  = #0x35
constant KEY_V  = #0x36
constant KEY_W  = #0x37
constant KEY_X  = #0x38
constant KEY_Y  = #0x39
constant KEY_Z  = #0x3A
constant KEY_0  = #0x10
constant KEY_1  = #0x11
constant KEY_2  = #0x12
constant KEY_3  = #0x13
constant KEY_4  = #0x14
constant KEY_5  = #0x15
constant KEY_6  = #0x16
constant KEY_7  = #0x17
constant KEY_8  = #0x18
constant KEY_9  = #0x19
constant KEY_SP = #0x02
```

```
.translate_scancode:
; KEY_A Check
lod #scancode
mov b, #KEY_A
add #0xFE4      ;-0x1C
reteq
; KEY_B Check
lod #scancode
mov b, #KEY_B
add #0xFCE      ;-0x32
reteq

; KEY_C Check
lod #scancode
mov b, #KEY_C
add #0xFDF      ;-0x21
reteq

; KEY_D Check
lod #scancode
mov b, #KEY_D
add #0xFDD      ;-0x23
reteq

; KEY_E Check
lod #scancode
mov b, #KEY_E
add #0xFDC      ;-0x24
reteq

; KEY_F Check
lod #scancode
mov b, #KEY_F
add #0xFD8      ;-0x2B
reteq

; KEY_G Check
lod #scancode
mov b, #KEY_G
add #0xFCC      ;-0x34
reteq

; KEY_H Check
lod #scancode
mov b, #KEY_H
add #0xFCD      ;-0x33
reteq
```

```
; KEY_I Check
lod #scancode
mov b, #KEY_I
add #0xFBD        ;-0x43
reteq

; KEY_J Check
lod #scancode
mov b, #KEY_J
add #0xFC5        ;-0x3B
reteq

; KEY_K Check
lod #scancode
mov b, #KEY_K
add #0xFBE        ;-0x42
reteq

; KEY_L Check
lod #scancode
mov b, #KEY_L
add #0xFB5        ;-0x4B
reteq

; KEY_M Check
lod #scancode
mov b, #KEY_M
add #0xFC6        ;-0x3A
reteq

; KEY_N Check
lod #scancode
mov b, #KEY_N
add #0xFCF        ;-0x31
reteq

; KEY_O Check
lod #scancode
mov b, #KEY_O
add #0xFBC        ;-0x44
reteq
```

```
; KEY_P Check
lod #scancode
mov b, #KEY_P
add #0xFB3        ;-0x4D
reteq

; KEY_Q Check
lod #scancode
mov b, #KEY_Q
add #0xFEB        ;-0x15
reteq

; KEY_R Check
lod #scancode
mov b, #KEY_R
add #0xFD3        ;-0x2D
reteq

; KEY_S Check
lod #scancode
mov b, #KEY_S
add #0xFE5        ;-0x1B
reteq

; KEY_T Check
lod #scancode
mov b, #KEY_T
add #0xFD4        ;-0x2C
reteq

; KEY_U Check
lod #scancode
mov b, #KEY_U
add #0xFC4        ;-0x3C
reteq

; KEY_V Check
lod #scancode
mov b, #KEY_V
add #0xFD6        ;-0x2A
reteq
```

```
; KEY_W Check
lod #scancode
mov b, #KEY_W
add #0xFE3        ;-0x1D
reteq

; KEY_X Check
lod #scancode
mov b, #KEY_X
add #0xFDE        ;-0x22
reteq

; KEY_Y Check
lod #scancode
mov b, #KEY_Y
add #0xFCB        ;-0x35
reteq

; KEY_Z Check
lod #scancode
mov b, #KEY_Z
add #0xFE6        ;-0x1A
reteq

; KEY_0 Check
lod #scancode
mov b, #KEY_0
add #0xFBB        ;-0x45
reteq

; KEY_1 Check
lod #scancode
mov b, #KEY_1
add #0xFEA        ;-0x16
reteq

; KEY_2 Check
lod #scancode
mov b, #KEY_2
add #0xFE2        ;-0x1E
reteq
```

```
; KEY_3 Check
lod #scancode
mov b, #KEY_3
add #0xFDA        ;-0x26
reteq

; KEY_4 Check
lod #scancode
mov b, #KEY_4
add #0xFDB        ;-0x25
reteq

; KEY_5 Check
lod #scancode
mov b, #KEY_5
add #0xFD2        ;-0x2E
reteq

; KEY_6 Check
lod #scancode
mov b, #KEY_6
add #0xFC3        ;-0x3D
reteq

; KEY_7 Check
lod #scancode
mov b, #KEY_7
add #0xFC3        ;-0x3D
reteq

; KEY_8 Check
lod #scancode
mov b, #KEY_8
add #0xFC2        ;-0x3E
reteq

; KEY_9 Check
lod #scancode
mov b, #KEY_9
add #0xFBA        ;-0x46
reteq
```

```
; KEY_SP Check
  lod       #scancode
  mov       b, #KEY_SP
  add       #0xFD7      ; -0x29
  reteq

; Otherwise just return '?' symbol.
  mov       b, #0x1F
  ret

.lbl: dpw hello_world_str = "HELLO WORLD"
  dw        0x123
```

Literatura

[1] Patterson D. A., Hennessy J. L. *Computer Organization & Design, 5th Edition* Elsevier, 2014. ISBN: 978-0-12-407726-3

[2] Gu Ch. *Building Embedded Systems: Programmable Hardware.* Apress, 2016. ISBN 978-1484219188

[3] Pedroni, V.A., *Circuit Design and Simulation with VHDL* MIT Press, 2010, ISBN 978-0262014335

[4] Volder, Jack E. *"The Birth of CORDIC", Journal of VLSI Signal Processing.* Hingham, MA, USA: Kluwer Academic Publishers. ISSN 0922-5773

[5] Volder, Jack E. *"The CORDIC Trigonometric Computing Technique", IRE Transactions on Electronic Computers. The Institute of Radio Engineers, Inc. (IRE)* September 1959

[6] *Personal System/2 Hardware Interface Technical Reference – Common Interfaces* IBM Corp. 1990

[7] Paul Gigliotti *Implementing Barrel Shifters Using Multipliers* August 17, 2004, Xilinx, Inc.

[8] Hu, X., R. Huber and S. Bass *"Expanding the Range of Convergence of the CORDIC Algorithm"*, IEEE Transactions on Computers

[9] *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators* XAPP 052, July 7, 1996, Xilinx, Inc.

[10] Kemeny, John G.; Kurtz, Thomas E. *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System (4th ed.)*, Hanover, N.H.: Dartmouth College Computation Center, January 1968

[11] *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, New York, 1985.

[12] Patera Adam, *FPGA Demonstrátor soft mikroprocesoru*, Praha 2015. Bakalářská práce. České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra mikroelektroniky