FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Diplomová práce

REALIZACE 8BITOVÉHO MIKROKONTROLERU

Bc. Josef Šebánek

Vedoucí práce: Ing. Pavel Lafata, Ph.D.
Katedra telekomunikační techniky FEL

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šebánek** Jméno: **Josef** Osobní číslo: **478259**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Elektronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace základního 8bitového mikrokontroleru v FPGA poli

Název diplomové práce anglicky:

Designing and Implementation of Basic 8bit Microcontroller into FPGA

Pokyny pro vypracování:

Navrhněte a v jazyce VHDL implementujte základní 8bitový mikrokontroler postavený na vhodné architektuře. Mikrokontroler musí obsahovat veškeré bloky potřebné k funkci, tedy řadič (řídicí jednotka, registry, aritmeticko-logickou jednotku (ALU), paměť typu ROM a paměťový řadič, případně rozšířený matematický koprocesor FPU nebo časovače. Implementujte navržený mikrokontroler do vhodného FPGA pole a vývojového kitu (např. DE10-Lite, Nexys4). Otestujte mikrokontroler a demonstrujte na daném vývojovém přípravku jeho jednotlivé funkce: provádění základních aritmetických a logických operací, matematických operací s plovoucí desetinnou čárkou, vykonávání cyklů, podmínek, operací s posuvnými registry a registry s funkcí rotace, práci s pamětí (adresace, ukládání, načítání) apod. Vytvořte v jazyce VHDL sadu vhodných ukázek pro demonstraci funkcí mikrokontroleru.

Seznam doporučené literatury:

[1] Pinker, J. - Poupa, M.: Číslicové systémy a jazyk VHDL. Praha : BEN - technická literatura, 2006. 349 s. ISBN 80-7300-198-5.
[2] Ashender, P., J.: The VHDL Cookbook [online]. Dostupné z: https://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Pavel Lafata, Ph.D. katedra telekomunikační techniky FEL

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

Datum zadání diplomové práce: **31.01.2023** Termín odevzdání diplomové práce: **26.05.2023**

Platnost zadání diplomové práce: **22.09.2024**

Ing. Pavel Lafata, Ph.D.	prof. Ing. Pavel Hazdra, CSc.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	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

PROHLÁŠENÍ

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Ve Vraném nad Vltavou, 20. března 2023

ANOTACE

Tato práce má za cíl vytvořit jednoduchý mikrokontroler s využitím kitu Nexys 4 založeného na technologii FPGA. V první části se seznamujeme s obecnými principy mikrokontrolerů a technologií FPGA. V praktické části popisujeme veškeré funkční bloky mikrokontroleru a jeho možnosti. V poslední části se zaměřujeme na programy, které ukazují všechny funkce vytvořeného mikrokontroleru.

Klíčová slova: FPGA, mikrokontroler, VHDL, Nexys 4, Artix 7

ANNOTATION

This thesis aims to create a simple microcontroller of using Nexys 4 development kit based on FPGA technology. The first part introduces microcontrollers in general and FPGA technology. The practical part describes all the functional blocks of the microcontroller and its capabilities. The last part focuses on the programs that show all the functions of the created microcontroller.

Keywords: FPGA, microcontroler, VHDL, Nexys 4, Artix 7

PODĚKOVÁNÍ

V prvé řadě bych chtěl poděkovat vedoucímu diplomové práce Ing. Pavel Lafata, Ph.D. za odborné vedení a cenné rady při vypracování této práce. Dále bych rád poděkoval celé mé rodině, která mě během studia podporovala.

OBSAH

OBRÁZKY

TABULKY

1 ÚVOD

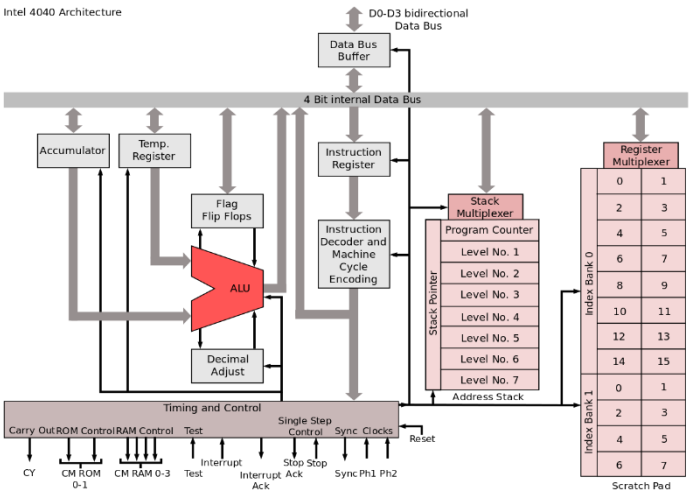
V současné době si nedokážeme představit život bez moderních technologií. Každou vteřinu se vyprodukuje miliony TB dat, které musí být zpracovány. K zpracování těchto dat jsou nejvíce využívány mikrokontroléry (MCU), které nejen zpracovávají data, ale také technologie celkově řídí. Proto již MCU lze najít téměř v každém zařízení od počítačů přes mobily, televize, fotoaparáty, ale také pračky, rychlovarné konvice, myčky nebo například kola. MCU jsou ale omezeny, svojí velikostí sběrnice a pracovní frekvencí. Pokud chceme zpracovávat data s větším datovým tokem je zapotřebí využít jiné technologie. Příkladem jsou zákaznické integrované obvody (ASIC, anglicky Application Specific Integrated Circuit), které jsou specializované pro danou problematiku. Nevýhodou takových obvodů je jejich výrobní cena při menších sériích. Pokud ovšem chceme integrovaný obvod, který zvládne velké množství dat a zároveň nízké náklady, nejlepší volbou jsou programovatelná hradlová pole (FPGA, anglicky Field Programmable Gate Array). FPGA umožňují velkou variabilitu ve zpracování velkého množství dat. Pokud je u daného výrobku potřeba například přejít na nový komunikační protokol, který využívá ještě navíc jiné rozložení sběrnice, tak u klasických MCU by bylo předělání složité. U FPGA v takovém případě stačí aktualizovat vnitřní zapojení a produkt může fungovat i nadále.

Pro fungování takového zařízení je také potřeba využít nějaký jednoduchý MCU k ovládání a řízení dalších komponent. To ovšem znamená, že je potřeba přidat další integrovaný obvod. V dnešní době je kvůli zmenšování snaha vše integrovat na jeden čip. Proto se v této práci budu zabývat vytvořením jednoduchého mikrokontroleru přímo na FPGA. Spousta výrobců již takové možnosti nabízí. Příkladem jsou jádra ARM. Nevýhodou je, že se tyto mikrokontrolery implementují jako funkční IP blok, který lze upravovat jen do určité míry. Můj cíl bylo vytvořit mikrokontroler ve formátu VHDL, který bude možné implementovat na libovolné FPGA s možností libovolné úpravy samotného MCU. [1]

2 MIKROKONTROLER

2.1 HISTORIE

Počátky mikrokontroléru sahají až do roku 1959, kdy byl vytvořen první tranzistor MOS. Tyto tranzistory díky menší spotřebě dosáhly v roce 1964 vyšší hustoty tranzistorů a nižších nákladů než technologie bipolárních tranzistorů. První mikroprocesor (Intel 4004), viz obrázek 2.1, vznikl až v roce 1971 a byl představen společně s celou rodinou 4000 společností Intel. Tento mikroprocesor byl pouze čtyřbitový a fungoval na frekvenci 750 kHz. V sérii 4000 byly představeny další podpůrné obvody, jako například 4001 – ROM paměť, 4002 – RAM paměť a 4003 – I/O posunovací registr. Instrukční sada tohoto mikroprocesoru obsahovala 45 instrukcí. Tento mikroprocesor byl původně vyvinut pro kalkulačky Busicom, ale kvůli velké míře využitelnosti se tyto mikroprocesory dostaly i do dalších zařízení. Čtyřbitový mikroprocesor má ovšem problém s malou datovou sběrnicí pro paměť RAM. Proto byl v roce 1972 vyvinut Intel 8008, který již byl osmibitový. Intel 8008 pracoval ze začátku pouze na frekvenci 500 kHz, která byla později zvýšena na 800 kHz. Kvůli možnosti zpracovat až osm bitů najednou byl až čtyřikrát rychlejší než jeho čtyřbitová konkurence.



Obrázek 2.1 Blokové schéma procesoru Intel 4004 [3, s.1]

Do té doby se vyráběly pouze mikroprocesory, které potřebovaly další obvody, jako RAM a ROM, k úplnému ovládání zařízení. To se ovšem změnilo roku 1976, kdy byla představena řada jednočipových mikropočítačů (mikrokontrolerů) MCS-48. Zástupci této řady byli 8048, 8035 a 8748. Tyto mikrokontroléry měly paměť o velikosti 1K x 8 ROM a 64 x 8 RAM. Jejich I/O se pohybovaly od 13 do 27. Model 8021 využíval například stolní počítač TRS-80 Model II, který tento mikrokontrolér implementoval do klávesnice, která se kvůli němu dala odjímat od zbytku počítače. Tato řada se používala díky své ceně a jednoduchosti až do roku 2000. [2]

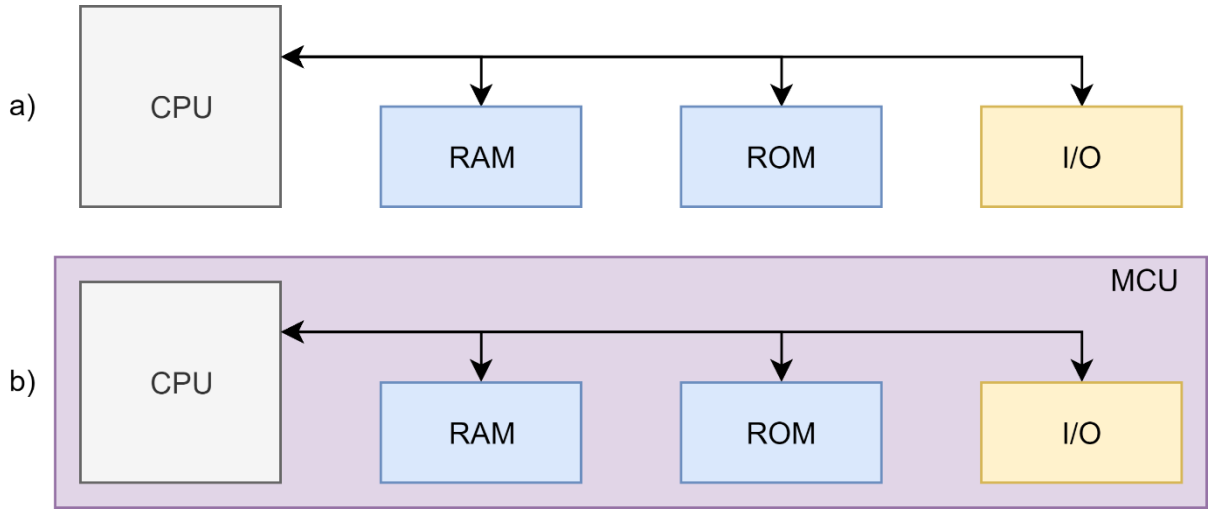
Dnes jsou největšími výrobci mikrokontrolerů ATMEL s (AVR), MICROCHIP (PIC) a MOTOROLA. Tyto mikrokontroléry prošly dlouhým vývojem, a proto již obsahují mnoho dalších komponent, jako jsou například A/D a D/A převodníky, interní časovače, PWM atd. Také obsahují různé druhy rozhraní, jako je USART, SPI, I2C, USB atd. Díky těmto blokům jsou schopny jednoduše a efektivně ovládat téměř jakékoliv zařízení.

2.2 ROZDÍL MIKROKONTROLERU A MIKROPROCESORU

Jelikož se v této práci často zmiňuji jak o procesorech (µP, CPU), tak o mikrokontrolerech (MCU), je důležité nejprve popsat rozdíly mezi nimi. Tyto rozdíly také vysvětlují, proč se tato práce zaměřuje na vytvoření mikrokontroleru.

Procesor (μP, CPU) obsahuje základní jednotky, jako jsou ALU, registry, řídící jednotka a datové sběrnice. CPU však nemůže fungovat samostatně, protože potřebuje další obvody: ROM, RAM a I/O porty. Procesory jsou například používány v osobních počítačích, kde je vidět jejich oddělení od portů a pamětí.

Mikrokontroler (MCU) v sobě obsahuje nejen samotný mikroprocesor (µP, CPU), ale také další obvody, jako jsou ROM, RAM a I/O porty. Díky tomu dokáže fungovat samostatně, bez potřeby dalších obvodů. I když je mikrokontroler menší než mnoho procesorů, může obsahovat širokou škálu pamětí a portů pro vstup a výstup. Kvůli své malé velikosti se dnes mikrokontrolery nacházejí téměř všude. Tento rozdíl je ukázán na obrázku 2.2. [3]



Obrázek 2.2 Rozdíl mezi CPU a MCU

2.3 DĚLENÍ MIKROKONTROLERŮ NA RISC A CISC

Během 70. let došlo při vývoji procesorů a mikrokontrolerů k postupnému rozšiřování instrukčních sad. To vedlo k větší složitosti a velikosti zapojení. Výhodou takových mikrokontrolerů je, že dokážou během jednoho cyklu vykonat složité úlohy. Nevýhodou jsou však velké energetické nároky a problémy s rozvodem hodinových signálů. Koncem 70. let se tento problém začal řešit rozdělením mikroprocesorů do tří hlavních skupin:

- CISC – Počítače se složitým souborem instrukcí. (Complex Instruction Set Computer)
- RISC – Počítače s redukovaným souborem instrukcí. (Reduced Instruction Set Computer)
- SISC – Počítače s jednou komplexní instrukcí. (Single Instruction Set Computer)

2.3.1 CISC

Mikroprocesory pracující na architektuře CISC mají integrované složité funkce, které by se daly rozdělit na více jednodušších instrukcí. Například násobení by mohlo být rozděleno na určitý počet součtů, které by však trvaly několik hodinových cyklů. S přímou implementací násobení do mikroprocesoru dokážeme dvě čísla vynásobit během jediného cyklu. Díky tomu existují instrukce, které zajišťují výpočet mocnin, odmocnin, goniometrických funkcí nebo třeba FFT. Ačkoli se zdá, že CPU pracující na architektuře CISC jsou schopny vykonávat vše rychleji, není to tak. Při vyšší taktovací frekvenci dochází k problémům s velkým množstvím logických hradel, kterými data procházejí. Tyto problémy se již projevují u komplexních funkcí a způsobují obrovské zpoždění, což ovlivňuje i taktovací frekvenci. [4]

2.3.2 RISC

Tento problém lze vyřešit právě technologií RISC. Jedná se o redukovanou sadu instrukcí. Ve výsledku to znamená, že se snažíme co nejvíce složitých instrukcí rozebrat na základní operace. Teoretické minimum, v počtu instrukcí, jsou čtyři. V minulosti existoval mikroprocesor, který měl pouze osm instrukcí, pomocí kterých se všechny ostatní daly poskládat. Velkou výhodou takovýchto CPU je jejich možnost velmi velkých taktovacích frekvencí. To je umožněno díky jednoduchosti samotného CPU. Kvůli jednoduchosti zapojení využívá CPU i mnohem menší plochu než procesory CISC. Velkou nevýhodou je, že velmi složité instrukce mohou zaplnit velké množství programové paměti, nebo zabrat příliš velké množství programových cyklů. [4]

2.3.3 SISC

Tato technologie je v podstatě založená na předešlé RISC. Jedná se ale o technologii dovedenou do extrému. SISC neboli Simple Instruction Set Computer, je založen na implementování pouze jediné instrukce. Tato instrukce je pouze INC A. Jednoduše řečeno se jedná o inkrementaci. Vzhledem k tomu, že obsahuje pouze tuto jednu instrukci, tak odpadá potřeba některých bloků, jako je například RAM. Další zvláštností je, že tento μP nepotřebuje ani žádný program, jelikož provádí stále stejnou instrukci dokola. Proto bych tento μP ani mikroprocesorem nenazval. [4]

2.4 DĚLENÍ PODLE PŘÍSTUPU K PAMĚTI

2.4.1 VON NEUMANNOVA ARCHITEKTURA

Při návrhu prvních počítačů vznikalo mnoho konceptů, na jakých by mohly první počítačové jednotky fungovat. Jedním z konceptů byla architektura, kterou popsal matematik John Von Neumann. Jedná se o architekturu, ve které veškeré funkční bloky sdílí jednu společnou datovou sběrnici, skrz kterou protékají veškerá data. Hlavním rozdílem této architektury oproti většině dnešních počítačů byl přístup k paměti. Von Neumannova architektura využívá pouze jednu sdílenou paměťovou jednotku, ve které se nachází jak program, tak samotná data. Tato architektura má jisté výhody, které dovolují, aby byl program přepisován při běhu. Problémem této architektury je ovšem rychlost. Ta je především omezena rychlostí paměti. To je z důvodu, že při načítání instrukce nemůže být paměť použita pro práci s daty. Tento problém řeší dnes rozšířenější Harvardská architektura. [5]

2.4.2 HARVARDSKÁ ARCHITEKTURA

Tato architektura odděluje programovou paměť od paměti pro data. To umožňuje větší rychlost, jelikož lze načítat zároveň další instrukci a zároveň zpracovávat a ukládat data. Oddělením těchto pamětí je zabráněn přepis programu v rámci běhu počítače. Tím je zamezeno neúmyslný zásah do programové paměti. Harvardská architektura se dnes používá téměř ve všech počítačích a mikrokontrolerech. [5]

2.5 DĚLENÍ DNES

Dělení procesorů na RISC a CISC již dnes v podstatě neplatí, jelikož se využívají kombinace mezi nimi v závislosti na použití daného CPU. Proto je dnes spíše dělíme na: CPU, DSP, FPU, GPU a APU.

2.5.1 CPU

Central Processor Unit, neboli hlavní procesorová jednotka, je základem jakéhokoliv digitálního systému. Jedná se o procesor, který řídí celý systém včetně dalších procesorů. To je také důvod, proč každý mikrokontroler také má svůj CPU.

2.5.2 DSP

[6] Digitální signálové procesory jsou speciálně postavené pro velmi rychlý přístup k datům a zpracováním analogových signálů, které jsou již převedeny do digitální podoby. DSP jsou stavěny na Harvardské architektuře, jelikož mají vlastní sběrnici pro data, což umožňuje mnohem rychlejší přístup. Další specialitou, je velmi rychlá násobička společně se součtem. Důvodem je využívaní transformací a filtračních metod přímo se signálem. Digitální signálové procesory můžeme dále také dělit podle principu práce s daty na:

- Procesory s celočíselnou aritmetikou
- Procesory s plovoucí řádovou čárkou
- Procesory s pevnou řádovou čárkou

2.5.3 FPU

[7] Floating-point unit nebo také někdy nazývaný jako matematický koprocesor (MCP). ALU, jako taková, v mnoha případech neumí pracovat s čísly s plovoucí desetinnou čárkou. Tento problém se dá obejít pomocí programové emulace, každopádně se tím můžeme dostat k přetížení procesoru. FPU je právě uzpůsobena pro práci s desetinnou plovoucí čárkou, a proto ušetří velké množství práce pro CPU. Více o této jednotce je uvedeno v kapitole 2.12.

2.5.4 GPU

Grafická procesorová jednotka je speciálně uzpůsobena pro práci s obrazem. V dnešní době, se jedná o procesory, které v sobě obsahují tisíce jednoduchých výpočetních jader, která umí pouze základní operace. Pro výpočet obrazu těchto pár funkcí stačí. Důvodem je, že se pixely jednotlivých snímků stále určují pouze pomocí

celých čísel, tudíž není nutné počítání s plovoucí desetinnou čárkou. Tyto procesory se dají využít pro velké množství operací najednou, například se používají pro numerické simulace, nebo těžení kryptoměn. [8]

2.5.5 APU

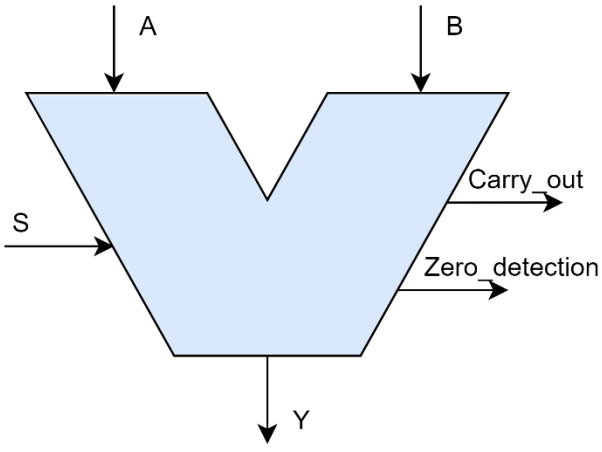
Accelerated Processing Unit spojuje dva důležité procesory (GPU, CPU) do jednoho čipu. Velkou výhodou tohoto spojení je mnohem větší datová propustnost mezi těmito jádry. To ve výsledku zvyšuje rychlost jak mobilů, tak osobních počítačů, které APU využívají. Dalším odvětvím, kde se APU používá, je průmysl herních konzolí

2.6 BLOKOVÉ SCHÉMA MIKROKONTROLERU

Protože existuje spousta různých druhů mikrokontrolerů, liší se i jejich blokové schéma. Proto se v této části podíváme na blokové schéma, které je obecné a bude přibližně platit pro většinu mikrokontrolerů. Toto blokové schéma obsahuje hlavní funkční bloky, bez kterých by mikrokontroler nemohl fungovat. Blokové schéma obsahuje jak bloky samotného mikrokontroleru, tak i bloky, které se vyskytují už v samotném CPU [9]

2.7 ALU

ALU (Aritmeticko-logická jednotka) se dá považovat za úplný základ všech CPU. Jak již z názvu vyplývá, jedná se o jednotku, která v závislosti na vstupní adrese S vykonává určitou operaci mezi dvěma vstupními sběrnicemi A a B. Výstupní hodnota operace je pak na sběrnici Y. Pokud by v průběhu operace došlo k překročení velikosti sběrnice, například při násobení, ALU vyšle signál do výstupu Carry out. Dalším výstupem u mnoha ALU jsou výsledky podmínek, ve kterých jsou porovnány vstupní hodnoty. Na obrázku 2.3 je znázorněn blok základní ALU s detekcí nuly.



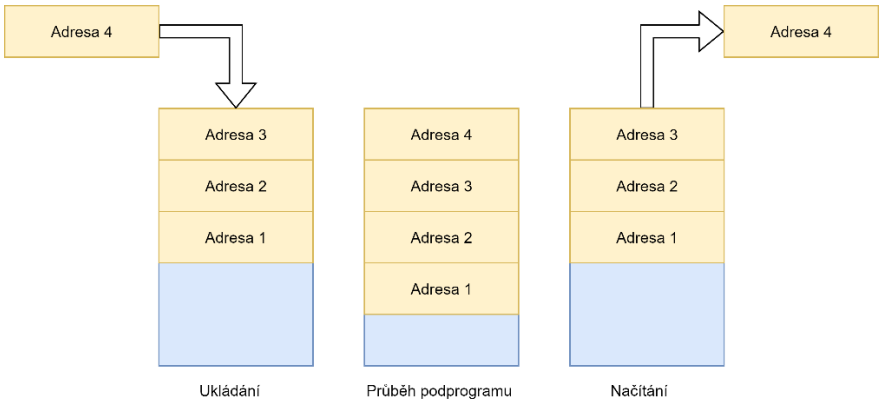
Obrázek 2.3 Blok základní ALU

2.8 PROGRAM COUNTER

Tento blok v sobě udržuje adresu aktuálně prováděné instrukce. Jedná se tedy o čítač adres. Tento čítač má také možnost nastavit adresu na určitou hodnotu. Této možnosti se využívá, pokud chceme skočit na určité místo v programu, nebo do podprogramu. Pokud chceme skočit do podprogramu, je potřeba další funkční blok (zásobník)

2.9 ZÁSOBNÍK

Zásobník (Stack) je malá paměť, kam se ukládá adresa, kde se program nachází, když vstoupí do podprogramu. To umožňuje programu po skončení podprogramu návrat na původní místo v kódu. Zásobník funguje na zásadě FILO (First In Last Out), což znamená, že adresa posledního vstupu je první adresa, která se vrací zpět. Díky této vlastnosti může program vstupovat do více vnořených podprogramů, protože adresa návratu se ukládá na zásobník a vrací se v opačném pořadí. Některé procesory mohou fungovat bez zásobníku, ale pak ztrácí schopnost vstupovat do podprogramů. Na obrázku 2.4. je ukázána funkce zásobníku.



Obrázek 2.4 Ukázka funkčnosti Stack zásobníku

2.10 PROGRAMOVÁ PAMĚT

Jak již z názvu vypovídá, jedná se o paměť samotného programu. Délka slova v paměti je ovlivněna počtem bitů MCU. To znamená, že při využití osmibitové architektury má každé slovo v paměti velikost osmi bitů. Pokud by ovšem velikost paměti byla omezena pouze osmibitovou sběrnicí, znamenalo by to, že dokážeme zapsat pouze $2^8 = 256$ slov. Proto je lepší využívat paměti, které mají větší velikost, alespoň tedy $2^{16} = 65536$ slov. Problém pak ovšem nastává při adresování takové velikosti, protože instrukce skoku mají velikost tří slov, kde první slovo je využité jako instrukce skok a další dvě slova jako adresa.

2.11 ŘÍDICÍ JEDNOTKA (CONTROL UNIT)

Samotným jádrem každého programově řízeného obvodu je řídící jednotka. Jedná se také o nejsložitější obvod v samotném zapojení procesoru (Složitostí není myšlena velikost). Hlavními vstupními signály do řídící jednotky jsou hodinový signál a výstup z programové paměti, tedy samotná instrukce. Řídící jednotka má za úkol v závislosti na instrukci ovládat veškeré prvky v CPU/MCU tak, aby byla instrukce správně provedena. Samotná instrukce může být provedena i ve více krocích. Proto je do samotného obvodu připojen i hodinový signál. To tedy znamená, že tato jednotka musí sama ovládat také skok na následující instrukci.

U některých procesorů je délka všech instrukcí stejně dlouhá, například čtyři cykly. To následně znamená, že každá instrukce bude trvat čtyři periody hodinového signálu. Takovéto zapojení zjednodušuje některé instrukce, jelikož má procesor více času na jednotlivé procesy. U jiných typů CPU má každá instrukce jinou délku. Tato varianta má jak výhody, tak negativa. Jednou z hlavních výhod je zrychlení výpočtů, jelikož se velmi jednoduché instrukce dokážou zredukovat na jeden cyklus. Nevýhodou ovšem je například složitější výpočet potřebného času pro vykonání programu.

2.12 FLOATING-POINT UNIT

FPU neboli Floating-Point Unit je jednotka, která byla již zmíněna v 2.5. Jejím úkolem je pracovat s čísly, která obsahují plovoucí desetinnou čárku. Díky tomu je tato jednotka schopna zpracovávat nejen desetinná čísla, ale i čísla s velkými hodnotami. Pro práci s takovými čísly je nutné je nějakým způsobem zapsat.

Uvažujme konkrétní příklad. Pokud použijeme osmibitovou strukturu a datové typy signed a unsigned, dosáhneme pouze hodnot uvedených v tabulka 2.1. Tyto hodnoty však nestačí pro vyjádření větších nebo desetinných čísel.

Tabulka 2.1 Rozsahy datových typů v 8bitové technologii

DATOVÝ TYP	NEJMENŠÍ ČÍSLO	NEJVĚTŠÍ ČÍSLO
SIGNED	-127	127
UNSIGNED	0	255
MINIFLOAT	-240	240

Pokud ovšem využijeme datový typ minifloat při normě IEEE 754, tedy typ float o šířce osmi bitů, dosáhneme možnosti zápisu čísel v rozsahu od mínus -240 až do 240 s možností zápisu desetinných čísel do hodnoty $\pm\,0{,}0039$. Je jasné, že do osmibitového čísla nelze vložit tak velký objem dat. Jedná se tedy spíše o jiné zobrazení určitých hodnot

v daném formátu. Číslo je rozděleno na tři části: znaménko, exponent a mantisu (část, která určuje hodnotu za desetinnou čárkou), viz tabulka 2.2.

Tabulka 2.2 Rozložení datového typu Minifloat

Sign (S)	Exponent (B)				mantisa (M)		
1	1	1	0	0	1	0	1

První část čísla (S) určuje znaménko, takže při hodnotě 1 bude výsledné číslo záporné. Tento bit funguje podobně jako u typu signed. Druhou částí (E) je exponent, který udává konečný výsledek v binárním tvaru.

Protože tento zápis umožňuje i zápis desetinných čísel, může být exponent i záporný. Pro zajištění tohoto zápisu je od exponentu odečten nejvyšší bit, v tomto případě osmý. Výsledek části exponentu lze vyjádřit pomocí rovnice 2.1 kde exp je výsledný exponent, B je hodnota získaná z části exponentu a EB je nejvyšší hodnota exponenciální části. Právě část exponentu určuje rozlišovací schopnost velkých nebo malých čísel. Při použití EB v polovině (v tomto případě osm) získáme stejné rozložení hodnot v desetinné i mocninné části. V některých případech však potřebujeme použít vyšší hodnoty exponentu než desetinná čísla. V takových případech můžeme odčítání vynechat a dostat se až na hodnotu ± 61440 V důsledku toho se však sníží rozlišovací schopnost desetinné části na $\pm 0{,}125$.

$$exp = \ B - EB \tag{2.1}$$

Poslední částí je mantisa. Tato část udává hodnotu za desetinnou čárkou, kterou se číslo bude násobit. Jednoduše řečeno, toto číslo udává hodnotu, která zůstala za desetinnou čárkou po odstranění exponentu. Toto číslo tedy lze zapsat podle rovnice 2.2, kde hodnota XXX odpovídá hodnotě výsledku M a m odpovídá hodnotě čísla pro výpočet skutečné hodnoty.

$$m = \ 1,XXX \tag{2.2}$$

Výsledná hodnota celého čísla lze vypočítat pomocí rovnice 2.3. Tato rovnice platí pouze pro zápis čísel v normálním tvaru.

$$N = (-1)^S \cdot (1,XXX) \cdot 2^{exp} \tag{2.3}$$

Pro názornější ukázku převedeme číslo $(11100101)_2$ ve formátu Minifloat podle IEEE 754. Nejprve se podíváme na část exponentu, která je $(1100)^2 = 12$ Podle rovnice tedy dojdeme k výsledku:

$$exp = 12 - 8 = 4$$

Další částí je mantisa, jejíž hodnota je $(101)_2$ Výpočet této hodnoty bude vypadat takto:

$$m = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1{,}625$$

Následně vypočítáme celé číslo podle rovnice 2.3:

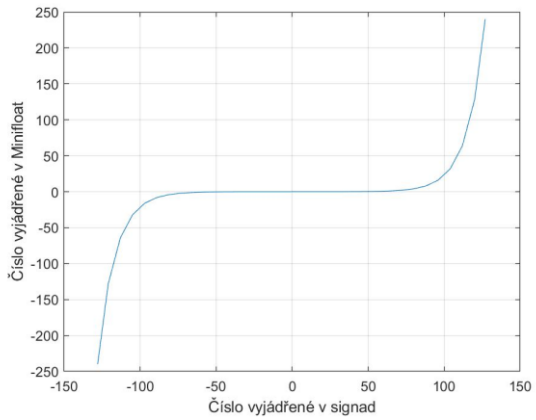
$$N = (-1)^S \cdot (m) \cdot 2^{exp} = (-1)^S \cdot (1{,}625) \cdot 2^4 = -26$$

Z toho tedy vyplývá, že výsledné číslo $(11100101)_2$ ve formátu Minifloat podle IEEE 754 má hodnotu -26.

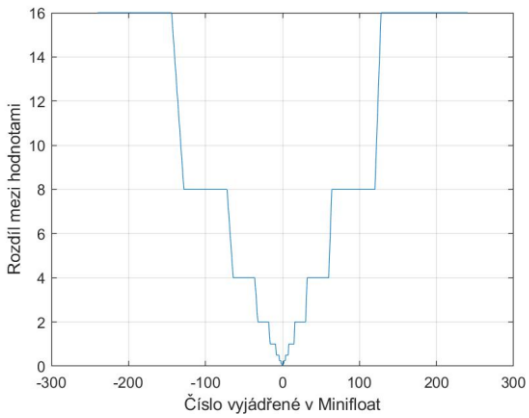
2.12.1 ROZSAH FLOAT

Jak již bylo zmíněno, tak tento zápis je pouze jinou interpretací dané hodnoty při stejné kapacitě. Z toho vyplývá, že nebude možné zapsat úplně každé požadované číslo. Proto při převádění do float dochází k převodu požadované hodnoty do nejbližší možné implementace.

Jako příklad si můžeme ukázat hodnotu 45. Pokud bychom se snažili zapsat tuto hodnotu pomocí minifloat, dojdeme k tomu, že to není možné. Tato hodnota tedy bude zapsána s jistou chybou, jako hodnota 44, která je nejblíže, a je vyjádřitelná jako $(01101011)_2$. Na obrázku 2.5 je graf rozložení hodnot minifloat oproti rozložení hodnot signed. Jak je vidět, čísla jsou vyjádřena exponenciálně. Čím více se tedy blížíme k vyšším číslům, tím dochází k větší chybě v zápisu čísel. Tato chyba je vyjádřena v grafu na obrázku 2.6 kde je vidět zvětšující se rozdíl, mezi dvěma následujícími čísly.



Obrázek 2.5 Graf rozložení čísel v Minifloat



Obrázek 2.6 Graf maximální chyby v interpretaci

2.12.2 OPERACE S FPU

Floating point unit umí základní čtyři operace: sčítání, odčítání, násobení a dělení. Vzhledem k způsobu zápisu se ale tyto operace provádějí jiným způsobem než v jednotce ALU pro celočíselné operace. Každá z operací se dá rozdělit do tří kroků:

1. Úprava čísel
2. Provedení operace
3. Normalizace

V prvním kroku se musí čísla převést do podoby, ve které lze operaci provést. Při sčítání a odčítání jde například o úpravu exponentu a mantisy. Tato úprava se provádí z důvodu, že čísla musí mít stejný exponent, aby se dala sečíst nebo odečíst. Pokud tedy máme čísla ve tvaru: $A = M_A \cdot 2^2$, $B = M_B \cdot 2^4$, je potřeba obě čísla převést do vyššího exponentu, tedy 4. Následně je potřeba u čísla A upravit mantisu tak, aby odpovídala upravenému exponentu. To se provádí pomocí bitového posunu. V tomto případě posuneme mantisu doprava o dva bity. Dále je potřeba číslo upravit podle znaménkového bitu. Pokud je číslo podle posledního bitu záporné, je potřeba mantisu převést do binárního tvaru záporného čísla. K tomu stačí provést jednoduchou operaci podle rovnice 2.4. Jde tedy o dvojkový doplněk (anglicky twos complement)

$$M\_ = not(M) + 1 \tag{2.4}$$

V druhém kroku probíhá samotná operace s čísly. Při sčítání a odčítání stačí v upravených číslech sečíst nebo odečíst pouze mantisy, jak už bylo zmíněno výše. Jelikož jsou již převedeny do znaménkového tvaru, výsledné číslo vyjde již se správným znaménkem.

U násobení a dělení je potřeba tuto operaci provést pro mantisy. Společně s tím se upravuje i exponent. Exponenty se při násobení sčítají a při dělení odčítají.

Posledním krokem je samotná normalizace čísel do podoby, ve které můžeme číslo opět zapsat v podobě Float. V tomto kroku dochází také k normalizaci. Vždy je potřeba výslednou mantisu převést do požadovaného tvaru: $1.XXX \cdot 2^e$ nebo případně u čísel $|M| < 1$ do tvaru $0.XXX \cdot 2^e$. To se provádí bitovým posunem. Společně s posunem mantis se také provádí korekce exponentu. [6]

2.12.3 OBECNÝ POPIS

Toto byla pouze jednoduchá ukázka toho, jak funguje floating-point unit. V dnešní době se samozřejmě hlavně používají formáty, které mají mnohem více bitů než pouhých osm. Typicky se v jednodušších mikrokontrolerech používá 16bitový formát half-float. Tato FPU je také součástí mikrokontroleru, který je popisován v této práci. Dalším formátem je již známý 32bitový formát float. Jeden z největších formátů pak následně obsahuje plných 64 bitů a nazývá se double. Veškeré rozložení těchto formátů je vidět v tabulce 2.3.

Tabulka 2.3 Velikosti datových formátů.

Formát	*Počet bitů sign.*	*Počet bitů exp.*	*Počet bitů mantis*
Mini-float	1	4	3
Half-float	1	5	10
float	1	8	23
double	1	11	52

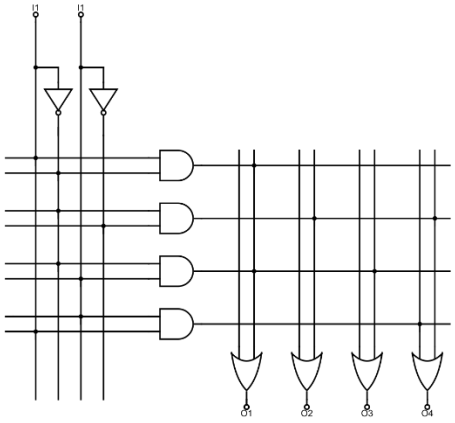
3 FPGA

Field Programmable Gate Array (FPGA) patří do celkové kategorie programovatelné logiky *Programmable Logic Devices* (PLD). Tyto obvody jsou obecně v dnešní době ve velkém vývoji kvůli své multifunkčnosti a rychlosti. Programovatelná logika již existuje dlouhou dobu, proto také můžeme najít mnoho různých typů, jako například: *Programmable Read Only Memory* (PROM), *Programmable Array Logic* (PAL) nebo FPGA.

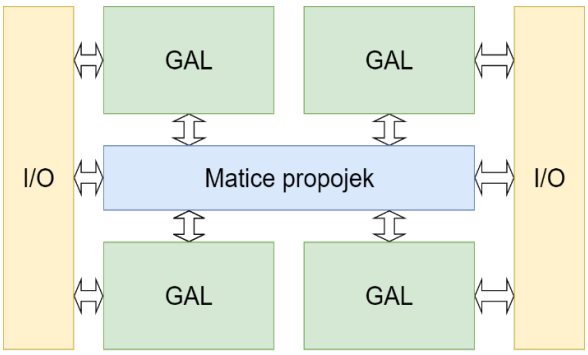
Důvodem, proč jsou dnes nejvíce využívané právě FPGA, je jejich vysoká multifunkčnost, jelikož lze pomocí nich nahradit téměř jakýkoliv čistě digitální obvod. K popisu zapojení vnitřní struktury FPGA využíváme speciální jazyky hardwarového popisu *Hardware Description Language* (HDL). Hlavními zástupci těchto jazyků jsou VHDL (VHSIC *Hardware Description Language – Very High-Speed Integrated Circuit Hardware Description Language*) a Verilog. Dalšími méně známými jazyky jsou například Verik, Ruby nebo MyHDL.

3.1 HISTORIE

První integrovaný obvod typu PLD byla paměť PROM, která byla vytvořena roku 1970. Jedná se o matici hradel AND a OR, viz obrázek 4.2 , které se programovaly přerušováním cest. Tyto obvody oproti klasickým logickým hradlům poskytovaly možnost vytvoření vlastní logické funkce až při stavbě konkrétního zapojení. S možností přeprogramování přišel až obvod GAL (Generic Array Logic), jehož struktura je stejná jako u obvodů PROM. Tyto obvody ovšem neumožňovaly vytváření složitějších logických obvodů. Z toho důvodu firma ALTERA vyvinula v roce 1988 obvod CPLD (Complex Programmable Logic Device). V principu se jedná o více propojených obvodů GAL pomocí propojovací matice, která tyto obvody následně propojuje s I/O porty. Tato struktura je ukázána na obrázku 3.1.



Obrázek 3.2 Struktura GAL [10, s. 4]



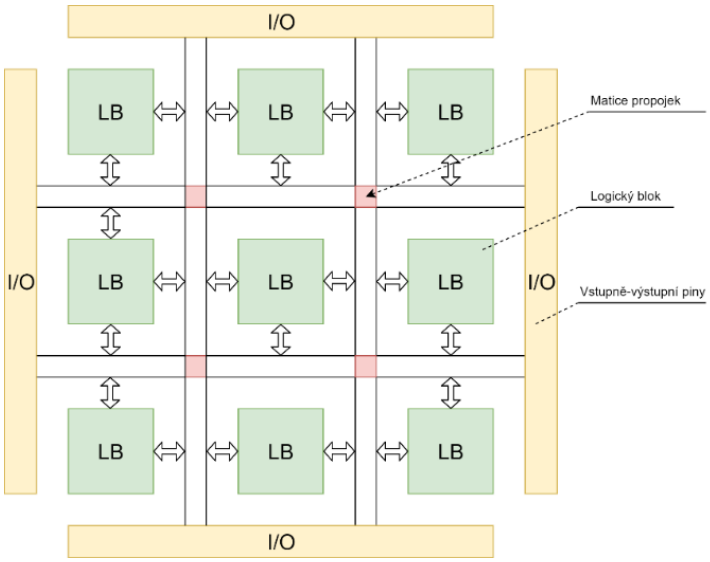
Obrázek 3.1 Blokové schéma CPLD [10, s. 9]

Zlatým obdobím pro vývoj programovatelné logiky bylo období 80 let. V této době firma MMI vyráběla sedm různých až dvaceti pinových obvodů typu PAL. Roku 1984 bylo firmou ALTERA vyvinuto první FPGA EP300, které dokázalo v sobě emulovat jakýkoliv ze zmíněných obvodů firmy MMI. EP300 bylo vytvořeno na bipolární technologii a obsahovalo v sobě 300 logických hradel v osmi makro buňkách. Velkou výhodou tohoto FPGA bylo jeho malé zpoždění mezi vstupem a výstupem, to dosahovalo maximálně okolo $90\ ns$.

Koncem 80. let již existovaly FPGA, které v sobě obsahovaly kolem 600 000 programovatelných hradel. Dalším vývojem byla implementace specifických funkčních bloků přímo do FPGA. To umožňovalo jednodušší implementaci specifických bloků, které se využívaly velmi často. Příkladem těchto bloků jsou: Paměti, násobičky, posuvné registry a mnoho dalších. To ve velké míře zefektivnilo využití hradlového pole a zásadně zrychlilo mnoho zapojení. Koncem 90. let se FPGA nejvíce uchytilo v telekomunikacích, kde umožnilo jednodušší přecházení na nové technologie nebo změny v síti.

3.2 ARCHITEKTURA FPGA

Struktura FPGA je založena na logických buňkách (LB) a horizontálních a vertikálních sběrnic, které jsou propojované pomocí matic propojek, viz obrázek 3.3 . Téměř veškerá kombinační i sekvenční logika je vytvářena pomocí LB.

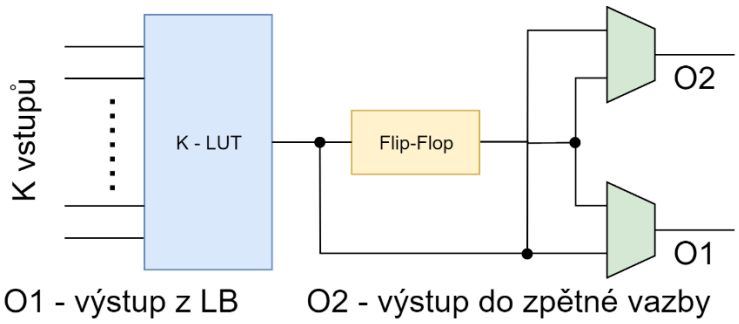


Obrázek 3.3 Struktura obecného FPG

Struktura těchto buněk se u různých výrobců může lišit, každopádně princip funkce je stejný. Dále jsou tyto buňky propojeny pomocí vertikálních a horizontálních sběrnic. Tyto sběrnice mohou v sobě obsahovat až stovky propojek. Pro propojení veškerých LB mezi sebou se starají Maticové propojky (Switch cell). Dalším blokem jsou pak vstupy a výstupy (*I/O*). Tyto buňky nejsou ničím odlišné od jiných obvodů se vstupně výstupními piny. Jedinou zvláštností je, že v sobě musí ukrývat další maticové propojky.

3.2.1 LOGICKÉ BLOKY (LB)

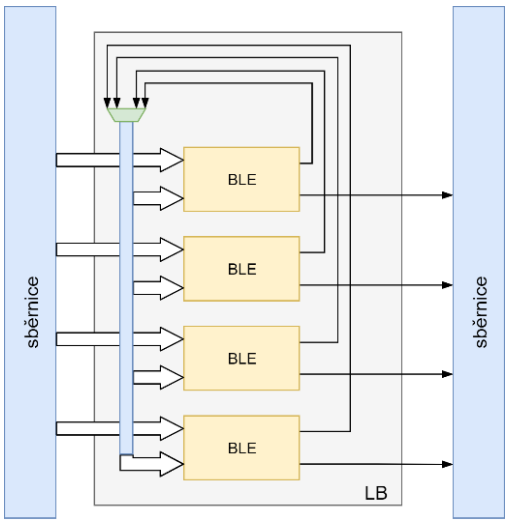
Logické bloky jsou hlavní součástí každého FPGA. Jak jsem již zmínil v předchozí části, tak pomocí právě těchto bloků je vytvořena veškerá logika. Struktura těchto buněk se už velmi liší jak mezi výrobci, tak mezi samotnými sériemi. Kromě toho se také ukázalo, že je výhodné, použít více druhů LB v jednom FPGA kvůli efektivnější implementaci různých zapojení. To je způsobené tím, že některé aplikace například potřebují větší datové sběrnice a některé naopak potřebují jen několik datových signálů s velkou přenosovou rychlostí.



Obrázek 3.4 Struktura BLE bloku

Obecně řečeno se logická jednotka vždy skládá z N bloků BLE (Základních logických elementů), které obsahují K vstupů. Klasicky se N pohybuje od dvou do osmi. Každopádně to nevylučuje větší počet. Každý BLE v sobě následně obsahuje LUT (Lookup table), několik multiplexorů a také Flip-Flop. Zapojení samotného BLE je vidět na obrázku 3.4 Struktura BLE bloku.

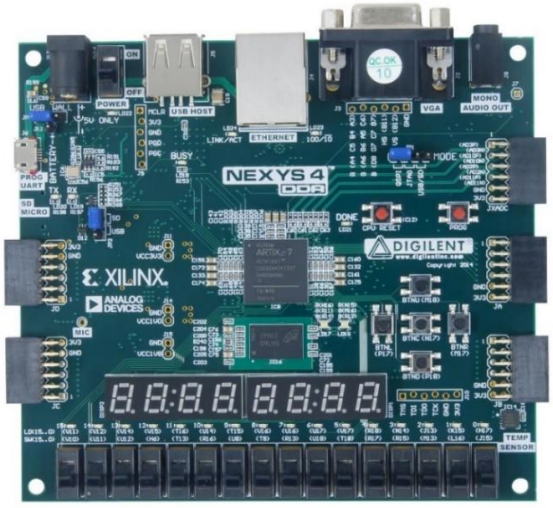
Veškerá základní kombinační logika, je vytvořena pomocí LUT. Ta zajišťuje veškeré možné kombinace v závislosti na K vstupech. Následný signál, je napojen do Flip-Flop, pomocí kterého můžeme vytvářet i veškerou sekvenční logiku. Výsledný signál je vyveden buď rovnou ven do sběrnic pomocí výstupu O1, nebo může být použit jako zpětnovazební signál, pomocí výstupu O2 vevnitř LB. Struktura celého LB je vidět na obrázku 3.5.



Obrázek 3.5 Struktura LB

4 VÝVOJOVÁ DESKA NEXYS 4

Pro tuto práci jsem si zvolil vývojovou desku Nexys4 od společnosti Digilent, která je postavena na FPGA od firmy Xilinx viz obrázek 4.1. Přesněji řečeno, využívá FPGA postavené na architektuře Artix 7. Důvodem pro použití této desky byla především možnost využití vývojového prostředí Vivado. Dalším důvodem je samotné FPGA, na kterém se dá vytvořit velmi komplexní systém díky mnoha využitelným blokům.



Obrázek 4.1 Digilent - Nexys 4

4.1 FPGA ARTIX 7

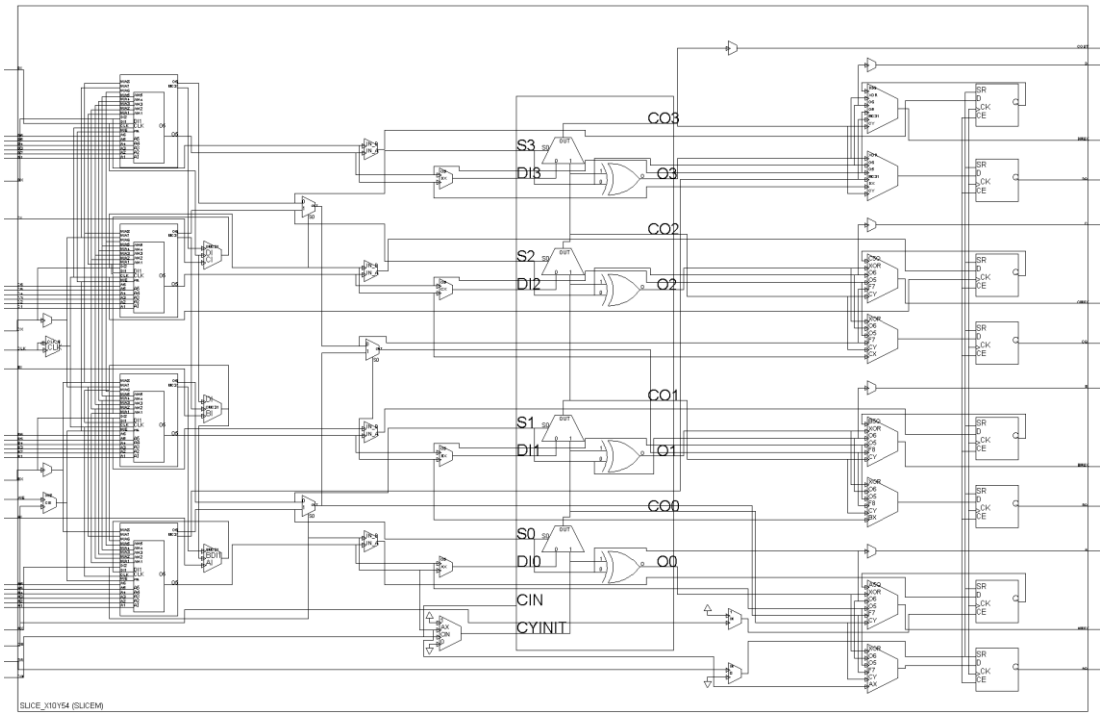
Jádrem celé desky je FPGA Artix XC7A100T-CSG324, které v sobě obsahuje 101 440 obyčejných logických buněk a dalších 15 850 konfigurovatelných logických buněk (CLBs). Tyto konfigurovatelné buňky v sobě obsahují dohromady až 1188 Kb distribuované paměti RAM. Další paměť pak zajišťuje 27018 Kb RAM, které dokáží pracovat jako 13536Kb bloky RAM. Dohromady tedy můžeme bez použití logických buněk vytvořit paměť RAM až s velikostí kolem 6000 Kb. Dalšími velmi užitečnými buňkami, které se v tomto FPGA nachází, jsou bloky DSP48E1, pomocí kterých se dají vytvářet rychlé násobičky a sčítačky. O spravování hodinových signálů se stará 6 Clock Managment Tiles (CMTs), kde každý z nich v sobě obsahuje jeden Phase-locked loop (PLL) a jeden Mixed-mode Clock Manager (MMCM). Do samotného čipu vstupuje hodinový signál o frekvenci 100 MHz, každopádně maximální frekvenci, na které lze FPGA provozovat je 450 MHz. Pro rychlou komunikaci je v FPGA blok který se stará o PCIe v2. Pomocí tohoto bloku lze dosáhnout komunikace s okolními periferiemi až o rychlosti 500 MB/s. Pro dosažení další možností rychlé komunikace je vestavěno osmi vysílacích bloků GTPs pomocí kterých lze vysílat data rychlostí 6.6 Gb/s. Pro analogové signály je toto FPGA vybaveno jedním blokem XADC s rozlišením 12 b o rychlosti 1 MS/s. Pro komunikaci

s okolními periferiemi má tento čip k dispozici 300 plně nastavitelných pinů rozdělených do šesti I/O bank. [12]

Z popisu tohoto FPGA je jasné, že to je ideální varianta pro testování mikrokontroleru, ve kterém se mnoho z těchto bloků hodí. Pro lepší přehled jsou tyto bloky popsány v následujících kapitolách.

4.1.1 LOGICKÉ BUŇKY

Jak je vidět na obrázku 4.2 zapojení logického elementu, které je vytaženo přímo z programu Vivado, se nacházejí **LE**, které obsahují čtyři **BLE**. Každý z těchto logických elementů má šest vstupů společně s dalšími ovládacími signály. Tyto BLE jsou vzájemně spojené, takže lze pomocí nich tvořit logiku, která bude mít dohromady 24 vstupních signálů. Tyto buňky jsou propojeny vertikálně směrem nahoru. Z toho plyne, že lze ovládat spodními vstupy pouze výstupy v horní části. Na výstupu každé buňky je připojen obvod flip-flop, pomocí kterého lze signály „uložit". Tyto paměťové bloky mají společné ovládání a hodinové signály. Tím je zajištěno, že celá buňka LE pracuje synchronně a nedochází tak k rozdílnému zpoždění na jednotlivých výstupních větvích. Na výstupu v horní části se nachází dva signály s názvem COUT a D. Tyto signály mají na levé dolní straně své protějšky, kterými se dá spojit více LE buněk dohromady. Tím lze vytvořit zapojení, které paralelně spojí tyto buňky k vytvoření logiky o více než 24 vstupech. Jistá nevýhoda je ovšem v distribuci signálu COUT do CIN, která funguje sériově. To může při spojení více LE dohromady tvořit logické hazardy, a tím zvyšovat dobu na ustálení. [13]

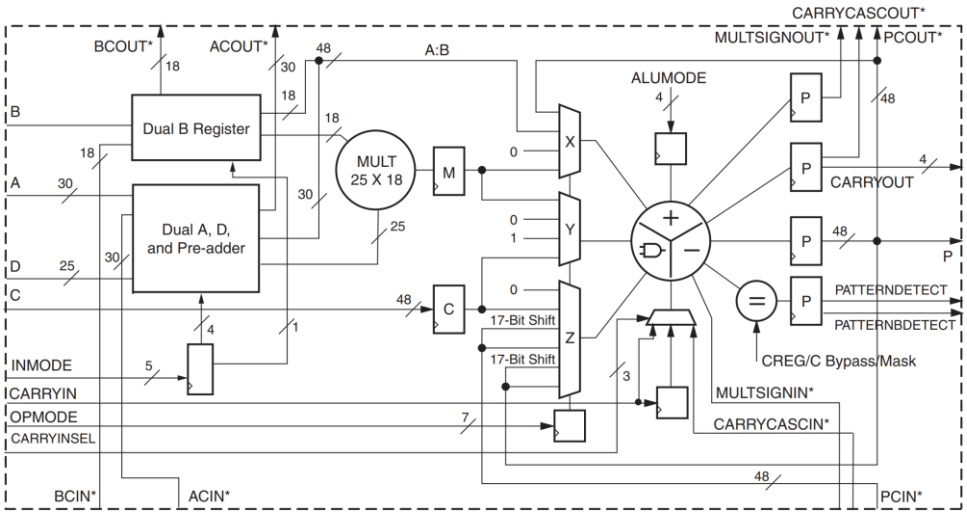


Obrázek 4.2 Zapojení logického elementu

4.1.2 BUŇKA DSP48E1 [14]

Jelikož jsou FPGA uzpůsobeny pro zpracování digitálního signálu, tak jedním z bloků, které byly přidány do tohoto čipu, je blok pro Digital Signal Processing (DSP). Tato buňka je uzpůsobena tak, aby s ní šla například jednoduše provádět Fourierova transformace diskrétní v čase (DTFT). To je vytvořeno pomocí implementace rychlých bloků pro sčítání a odčítání a následně bloků pro násobení.

Tento blok, viz obrázek 4.3, má čtyři vstupní datové sběrnice A-D a jednu hlavní výstupní sběrnici P. Ostatní vstupy a výstupy slouží hlavně jako ovládání, signalizace nebo propojení s další buňkou pro DSP. Veškerá vstupní data A-D se nejprve kvůli synchronizaci ukládají do vstupních registrů. Sběrnice A a D jsou nejprve zpracovány v bloku Pre-adder. Jednoduše řečeno jde o vstupní sčítání nebo odčítání. Výsledek následně vstupuje společně s daty A do násobičky (MULT), která má velikost 25x18 b. Výsledná data pak pokračují do jednoduché ALU přes multiplexory X a Y. Tyto multiplexory slouží pouze pro přepínání mezi hodnotou jedna, nula nebo daty z MULT. Společně také do této ALU vstupují další data z multiplexoru Z. Tento multiplexor přepíná mezi datovým vstupem C, hodnotou nula, nebo také mezi daty z výstupu předchozí operace. Další data, která vstupují do Z, jsou data PCIN. Tato data přicházejí z jiného bloku DSP, přesněji z datového výstupu PCOUT, který je na horní straně. Výsledné hodnoty z ALU vedou společně s CARRYOUT přes registr na výstup nebo zpátky do ALU. Jelikož veškeré bloky jsou stavěné k tomu, aby umožňovaly variabilní délku datových sběrnic, tak i tento blok je opatřen vstupními a výstupními datovými sběrnicemi, které zajišťují propojení s více stejnými bloky. Tyto sběrnice jsou vidět na obrázku 4.3 v dolní a horní části, přesněji jde o sběrnici BCOUT, která se propojuje v BCIN. Stejně tak to platí pro AC, PC, MULTISIGN nebo CARRYCASC. Samotná tato jednotka dokáže pracovat až na rychlostech, na které je uzpůsobeno samotné FPGA, přesněji 450 MHz. Pro tuto práci ovšem budou stačit samotné bloky bez propojení, jelikož se v ní nevyskytují větší operace než šestnáctibitové.

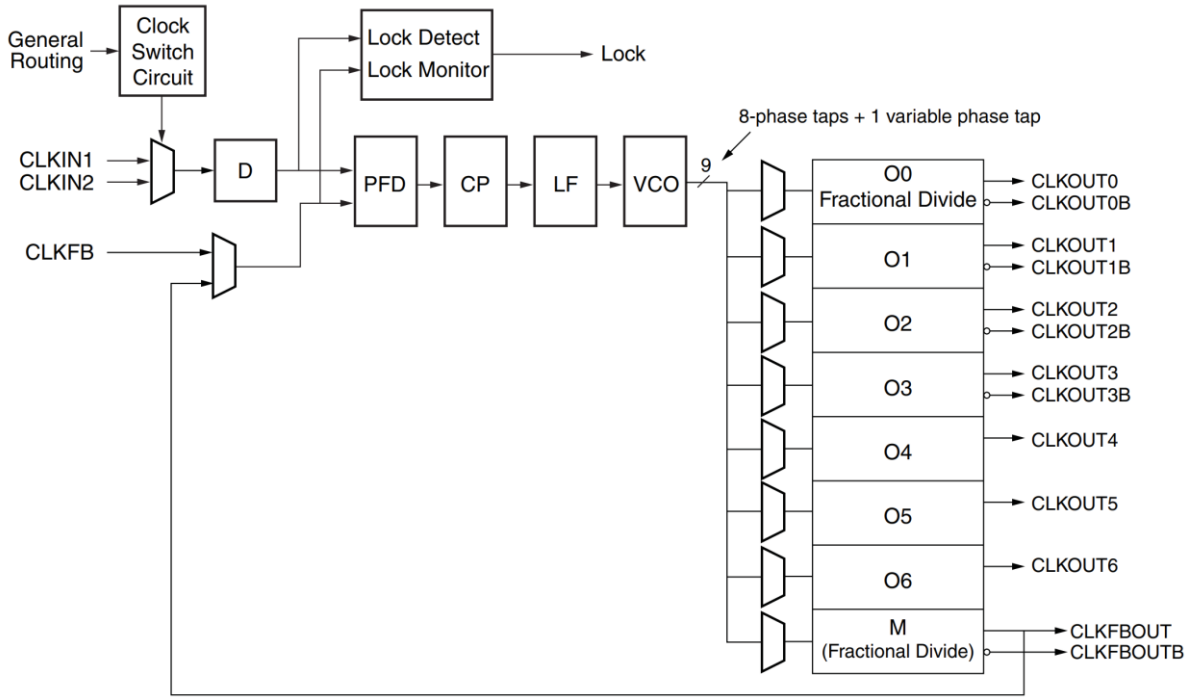


Obrázek 4.3 Struktura bloku DSP48E1 [14, s.14]

4.1.3 PLL A MMCM

Základem každé sekvenční logiky je správa hodinových signálů. Je jasné, že kdybychom měli do FPGA přivádět veškeré hodinové signály, které potřebujeme, tak bychom pro složitější aplikace spotřebovali mnoho vstupů jen pro hodinové signály, nemluvě o logice, která by musela být kolem a zajišťovala by jejich tvorbu. Proto je v dnešní době téměř standardem, dávat do FPGA polí Phase-locked Loop (PLL). Pomocí PLL lze vytvářet téměř jakékoliv hodinové signály z libovolných vstupních signálů. Jedná se totiž o digitálně analogový obvod se zpětnou vazbou, který v sobě obsahuje napětím řízený oscilátor. Menší nevýhodou tohoto bloku je doba ustálení. Jelikož obvod funguje zpětnovazebně, je potřeba zajistit nějaký čas na ustálení tohoto signálu. Pro ustálení tak Lock Monitor přepne signál Lock na logickou jedničku, a tím informuje, že jsou již hodiny použitelné.

To ovšem stále neřeší více požadovaných hodinových signálů. K tomu firma Xilinx přidala do PLL blok Mixed-mode Clock Manager (MMCM), který umožňuje vytvořit dohromady až šest různých hodinových taktů, které zároveň mezi sebou nemají fázovou odchylku. Dále také tvoří k těmto signálům jejich inverzní varianty. Celé blokové schéma tohoto bloku je vidět na obrázku 4.4, kde v první části jsou vidět samotné bloky PLL, přesněji bloky D, PFD, CP, LF a VCO společně s detekcí ustálení, a následně je připojen blok MMCM, který umožňuje tvoření dalších hodinových signálů pomocí nastavitelných děliček.



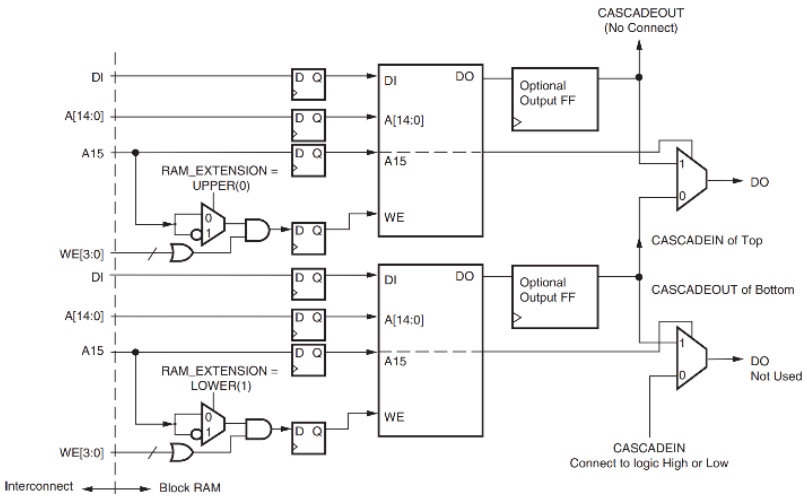
Obrázek 4.4 blokové schéma PLL a MMCM [15, s.67]

Pro distribuci globálních hodinových signálů jsou v FPGA vytvořeny speciální maticové spoje, které mají minimální zpoždění. Pomocí těchto spojů tak lze rozvést hodiny od PLL do jakékoliv části hradlového pole bez potřeby zbytečných zpoždění v cestě. Tím je zajištěno, že veškeré hodinové signály budou mít téměř totožnou fázi v každé části pole. [15]

4.1.4 PAMĚŤ RAM

Pro mnoho aplikací je nedílnou součástí nějaký paměťový modul. To bylo dříve pro FPGA velký problém, jelikož tvoření paměťových modulů pomocí LE je velmi neefektivní, co se týká využití hradel a rychlosti. Proto se do dnešních FPGA přidávají bloky RAM pamětí, které tento problém velmi dobře řeší. V tomto konkrétním čipu je využito hned několik druhů pamětí, které budou popsány v této kapitole.

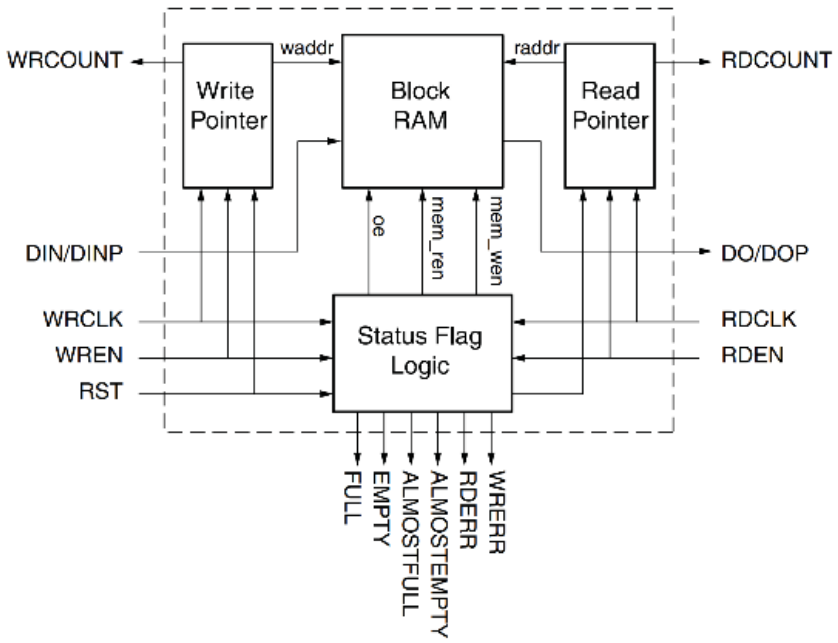
Jak již bylo napsáno na začátku této kapitoly, tak hlavní RAM paměti lze využívat ve dvou módech. V jednom módu pracuje paměťový blok jako jedna RAM paměť o velikosti 64 kb, neboli 64b data s 16b adresou. Ve druhém módu se paměť dělí na dvě nezávislé paměti o velikosti 32 kb, neboli 32b data s 16b adresou. Tím je zajištěno, že při menších datech nevyplýtváme celou paměť. Celá buňka je opět opatřena hodinovým signálem a vstupními registry kvůli synchronizaci. Jak je vidět na obrázku 4.6, tak je paměťový blok také doplněn o propojovací vstupy a výstupy CASCADE, které umožnují práci více bloků paralelně. Tím je pak možné vytvářet paměťové moduly pro data o větší velikosti.



Obrázek 4.5 Blokové schéma bloku pro RAM [16, s.23]

Dalším paměťovým blokem, který je přímo implementován v čipu, jsou paměti typu FIFO. Jednoduše se dají tyto obvody popsat jako zásobníky pro uložení dat před zpracováním. Jak je vidět na obrázku 4.5, tak tato paměť funguje na principu paměti RAM, na kterou jsou napojeny vstupní a výstupní ukazatele. Dále tato paměť využívá status Flag logic, který zajišťuje kontrolu přetečení. Tak lze jednoduše zjistit, že je již paměť plná

a nedají se ukládat další data. Tyto FIFO paměti jsou opět konstruovány tak, aby bylo jednoduché je spojovat s dalšími FIFO. Každý z těchto bloků dokáže v sobě uchovat až 512 hodnot o velikosti 144 b. Pro menší aplikace ho lze ovšem také rozdělit stejně jako u paměťových bloků do dvou nezávislých FIFO s datovou sběrnicí o velikosti 72 b.



Obrázek 4.6 Blokové schéma bloku FIFO

Poslední druh pamětí jsou samotné paměti nacházející se v LE. Tyto paměti jsou vhodné pro velmi malé paměťové bloky o několika bitech. Jedná se spíše o využití look-up tabulky jako paměťového bloku pro vytvoření paměti ROM nebo využití Flip-flop obvodu pro vytvoření paměti RAM.

5 VÝVOJOVÉ PROSTŘEDÍ VIVADO

Pro vývoj aplikací na FPGA pole existuje řada programů. Některé z těchto programů jsou tvořeny tak, aby pomocí nich šly tvořit aplikace na velké množství FPGA od různých menších firem. Větší společnosti, jako je Xilinx nebo Intel, vytvořily vlastní nástroje, které jsou uzpůsobeny především pro tvorbu aplikací na jejich vlastní FPGA. Tím je zajištěna vysoká kompatibilita a maximální vytěžení všech dostupných bloků v poli. Příkladem je vývojové prostředí Vivado, které pracuje s novějšími FPGA od firmy Xilinx. Jelikož Xilinx má své výrobky na trhu mnoho let, tak Vivado je nástupcem jejich předchozího vývojového prostředí ISE, které se ještě dnes využívá pro vývoj na starších čipech.

Jak již bylo napsáno v kapitole 4 , tak Vivado bylo jedním z důvodů, proč jsem si vybral pro tuto práci právě desku s čipem od firmy Xilinx. Hlavním důvodem je to, že umožňuje jednoduchou správu veškerých modulů a velké množství užitečných funkcí.

V programu lze tvořit aplikace pomocí několika různých možností. Buď čistě pomocí HDL, jak VHDL, tak Verilogu, nebo pomocí kombinace s blokovým editorem. Velkou výhodou je, že pro zapojení, vytvořené pomocí blokového editoru, můžeme následně vytvořit popis pomocí HDL.

Velkou výhodou tohoto prostředí, oproti například Quartus Prime od firmy Intel, je jednoduché měnění Top level entity. To je velmi užitečné, když je potřeba testovat a upravovat jen jednu část z celého zapojení. V tomto programu lze změnit hlavní entitu téměř jedním kliknutím. To ušetří velké množství času. Veškeré soubory jsou ve Vivado rozděleny do tří základních sekcí:

- Disign Source – Pro veškeré bloky, které budou přímo implementovány.
- Constraints – Soubory pro nastavování FPGA, jako jsou vstupy, výstupy nebo rychlosti.
- Simulation Sources – soubory sloužící jako test bench v simulacích.

5.1 SIMULACE

Právě simulace jsou jedním z velmi silných nástrojů, které ve Vivadu jsou dotažené téměř k dokonalosti. Spousta programů, se kterými jsem se setkal, obsahuje především dva druhy simulací: Behavioral a Implementation. Tyto simulace jsou sice velmi užitečné, ale není tak snadné odhalit, v jaké části se vyskytují konkrétní chyby. Vivado obsahuje rovnou pět druhů simulací.

5.1.1 BEHAVIORAL SIMULATION

Tato simulace je jedna z prvních simulací, která se dá použít hned po uložení modulu. Testuje pouze napsaný kód bez jakékoliv syntézy a implementace. Tím lze nalézt chyby, které nastaly už při samotném návrhu dané aplikace. Tato simulace využívá pro svůj chod veškeré výchozí hodnoty všech signálů, což znamená, že můžeme testovat dané zapojení při různých situacích. Další výhodou v této simulaci je použití takzvaných break pointů. Ty jsou podobné jako například při programování v jazyce C. Pokud tedy testujeme sekvenční obvod, který obsahuje podmínky, můžeme pomocí break pointu zjistit, jaké mají hodnoty jednotlivé signály v danou chvíli.

5.1.2 POST SYNTHESIS

Prvním krokem při zpracování HDL jazyka je syntéza, po tomto kroku se veškerý kód přetvoří do schématu, které se skládá již ze známých bloků pro implementaci. Tyto bloky jsou zatím jen funkční celky, které bude později již jednoduché implementovat. Jelikož je předchozí zapojení proměněno v tyto bloky, může dojít k menším neočekávaným změnám. To se především stává, pokud nejsou definovány veškeré hodnoty určitých signálů při různých podmínkách. K odhalení těchto chyb slouží právě tento druh simulace. Tato simulace se ovšem dělí na další dva druhy. První simulace se zabývá pouze funkčností daného zapojení, a druhá k funkčnosti přidává zpoždění na jednotlivých blocích. Toto zpoždění je generované pouze jako minimální zpoždění daného bloku. Proto nelze počítat s tím, že výsledné zpoždění po implementaci bude stejné. Ukáže to alespoň přibližný pohled na to, jestli vzhledem k těmto zpožděním je toto zapojení vůbec schopno pracovat správně. Pokud se budeme chtít podívat na jednotlivé signály, můžeme si všimnout, že se zde objevily některé další signály nebo registry. To je způsobeno právě danou syntézou, která pro některé signály může vytvořit jejich registry nebo ovládací signály. To může být občas také znamení, že některé hodnoty nejsou přímo definovány.

5.1.3 POST IMPLEMENTATION

Posledním typem je simulace, která přímo simuluje dané zapojení už na samotném hradlovém poli. Pro tuto simulaci je důležité mít zvolené již konkrétní FPGA, na kterém bude implementováno zapojení. Implementace rozebere jednotlivé funkční bloky po syntéze na samotné reálné bloky v FPGA a implementuje dané zapojení do pole. To může přinést ovšem další řadu problémů. V tomto poli jsou před nastavením veškeré signály v neznámém stavu, což může způsobit nestabilitu obvodu. Dalším problémem, který může nastat, je ignorování jednotlivých bloků, které se zdají nepoužity. To následně způsobuje nesprávnou funkčnost zapojení. Dalším, co se může projevit, jsou již zmíněné nenastavené hodnoty jednotlivých signálů.

Jelikož se implementace snaží využít pole co nejefektivněji, mohou se tyto špatně nastavené hodnoty projevit až v tomto kroku. Tento druh simulace opět má dva další druhy, jako to bylo u Post synthesis. Opět lze testovat samotné zapojení nebo zapojení s veškerými zpožděními. Tato zpoždění jsou tentokrát brána jako zpoždění na veškerých hradlech. Proto je výsledná hodnota už velmi přesná. Simulace již ukazuje reálný pohled na danou implementaci. Proto by se výsledek již neměl lišit od skutečnosti. Pokud se podíváme do veškerých signálů, které tuto implementaci tvoří, tak můžeme vidět mnoho dalších vytvořených signálů, které nebyly předtím vytvořeny. To je právě tím, že jsou zde ukázány veškeré signály a sběrnice pro dané zapojení.

Samotné simulování celého obvodu je velmi výpočetně náročné, a tak může pro některé větší aplikace trvat i několik hodin. Tuto náročnost především způsobuje simulování veškerých hradel, kterých mohou být až desetitisíce. Pokud tedy simulaci pustíme, tak může probíhat až po jednotlivých pikosekundách, ve kterých bude vykreslovat jednotlivé nestabilní stavy veškerých signálů a sběrnic.

5.2 ZOBRAZENÍ ZAPOJENÍ

Jelikož tvorba zapojení pomocí HDL občas není úplně přehledné, nabízí Vivado možnost zobrazení daného zapojení pomocí schématu. To může velmi pomoci při hledání chyb, jelikož okamžitě ukazuje, jak byl kód pochopen a rozpoznán. Toto prostředí nabízí tři možnosti zobrazení.

První možností je zobrazení zapojení po analýze obvodu. Na tomto zapojení je vidět implementace pomocí známých funkčních bloků, jako jsou multiplexory, sčítačky, násobičky nebo registry. Toto zobrazení je přehledné a dá se jeho pomocí najít řada chyb. Nejčastější chybou, která se dá odhalit právě pomocí tohoto zobrazení, je ignorování celých funkčních celků, které analýza vyhodnotila jako nepoužívané.

Druhým zobrazením je zobrazení po syntéze. Toto zobrazení již není tak přehledné jako předchozí varianta, jelikož jsou veškeré bloky rozloženy do menších funkčních celků, ze kterých se tyto bloky dají složit. Toto zobrazení slouží především ke kontrole, zda syntéza nevyřadila další funkční celky nebo neodpojila nepoužívané signály. V komplexnějších systémech se více již nalézt nedá.

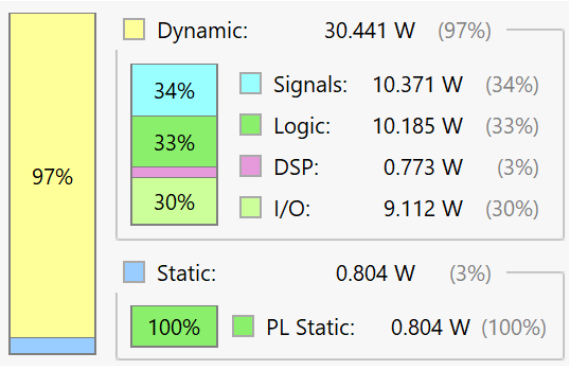
Posledním zobrazením je již samotná implementace, ta se od předchozích liší tím, že se jedná o zobrazení samotného FPGA a využitých bloků v něm. Toto zobrazení vypadá zajímavě, ale pro člověka je již skoro nemožné se v jednotlivých propojení vyznat. Hlavním důvodem, k čemu slouží, je kontrola umístění jednotlivých funkčních bloků.

5.3 INFORMACE O IMPLEMENTACI

Většina programů dokáže ukázat jen základní informace o dané implementaci, jako je například počet využitých LE a portů. Jelikož je Vivado stavěné pro návrhy i velmi složitých zapojení, byly do programu vloženy i další zajímavé ukazatele.

Po spuštění implementace dostaneme k dispozici okno Project Summary, ve kterém jsou vidět veškeré informace, které potřebujeme. Základním ukazatelem je již zmíněné využití hradlového pole. To jednoduše shrnuje, kolik bylo vypotřebováno jednotlivých bloků. Veškeré bloky jsou rozepsány do skupin: LUT, LUTRAM, FF, DSP, IO a BUFG. To nám dává jednoduchý přehled o tom, zda jsme využili veškeré bloky, které jsme pro danou implementaci potřebovali. Dalším ukazatelem, který je vhodný pro kontrolu rychlosti, je Timing. Ten naopak ukazuje celkové zpoždění daného bloku. Pro bližší informace si můžeme otevřít celou záložku týkající se veškerých časových reportů.

Dalším velmi užitečným nástrojem, který Vivado nabízí, je kontrola spotřeby na čipu a jeho teploty. Pokud tedy potřebujeme tvořit náročnější implementace, tento nástroj dokáže varovat, že by se čip mohl začít přehřívat. Případně také ukáže, na čem vznikají největší ztráty viz obrázek 5.1 Ukazatel spotřeby ve Vivado.



Obrázek 5.1 Ukazatel spotřeby ve Vivado

V neposlední řadě má Vivado skvělou kontrolu různých druhů chyb. Chyby se rozdělují do tří hlavních kategorií: chyby, kritická varování a varování. Veškeré chyby musí být odstraněny, jinak nemůže být zapojení implementováno. Nejčastěji se jedná o dvojité nastavování jednoho signálu nebo kód, který nelze implementovat.

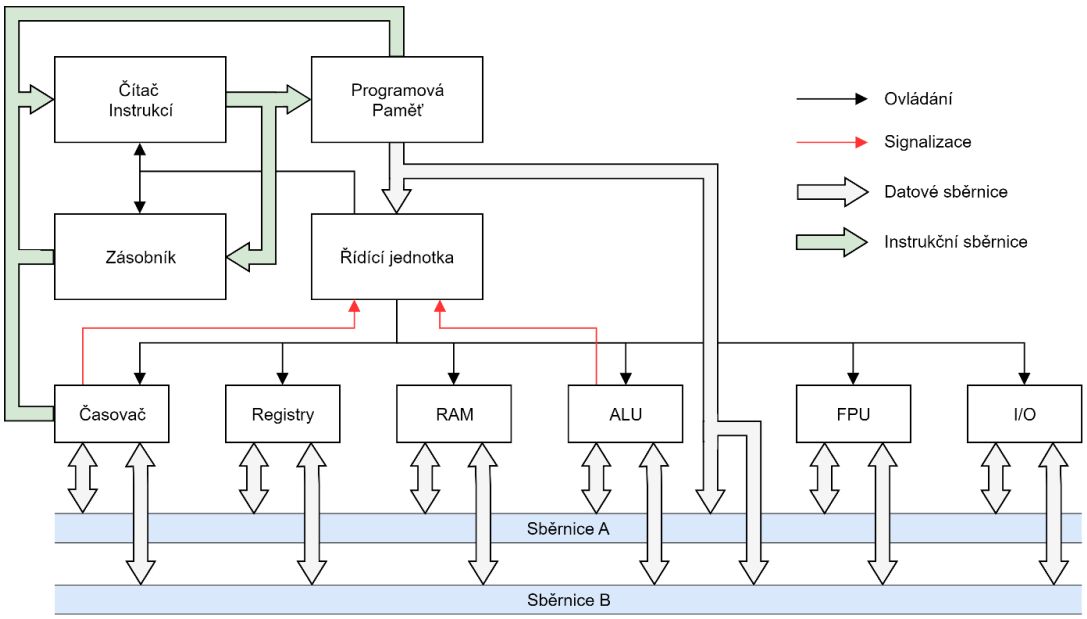
Kritická varování lze sice ignorovat a zapojení je implementovatelné, ale výsledný obvod nemusí fungovat tak, jak bychom si mysleli. Jedná se především o signály, na které má něco reagovat, ale nejsou vloženy do citlivostního seznamu. Další možností je špatně nastavení samotného čipu, které může způsobit odpojení některých signálů od I/O na čipu. Varování slouží především k menším chybám, které mohou způsobit různé funkce obvodu. Jedná se především o varování při vytvoření nečekaných bufferů nebo latch.

6 TVORBA MIKROKONTROLERU

Jak již bylo psáno v úvodní kapitole o mikrokontrolerech a mikroprocesorech, jedná se o celkově komplexní obvod, který zpracovává vstupní instrukce a provádí přesné operace. Tento mikrokontroler má být osmibitový, což znamená, že instrukce a data budou ukládána a zpracovávána pouze po osmi bitech. To ovšem velmi omezuje adresaci, která obsahuje pouze $2^8 = 256$ možných variant.

Pro vytvoření takto komplikovaného obvodu bylo nejprve potřeba vytvořit blokové schéma, podle kterého se následně mikrokontroler postaví. Blokových schémat mikrokontrolerů je k dispozici v různých literaturách mnoho. Většina z těchto schémat jsou ale hotové mikrokontrolery s předem určenými funkcemi. Můj cíl byl ale malinko odlišný. Hlavním cílem, který jsem si předem určil, byla větší variabilita a rozšiřitelnost samotného MCU. Z toho vyplývá, že z těchto schémat byl využit jen základní model, který je pro většinu MCU stejný, a zbytek byl upraven.

Vzhledem k rozšiřitelnosti bylo tedy potřeba vytvořit takové zapojení, ve kterém bude možné posílat veškerá data z jakékoliv části MCU do jakéhokoliv bloku. Pro tento účel jsem tedy přistoupil k variantě, kdy veškeré bloky sdílejí dvě hlavní osmibitové datové sběrnice A a B. Skrze tyto sběrnice pak procházejí veškerá zpracovávaná data. Jelikož jsou tyto sběrnice zavedeny do všech funkčních bloků, může tyto data zpracovat jakýkoliv blok.



Obrázek 6.1 Blokové schéma vytvořeného MCU

Jediná výjimka jsou adresy instrukcí. Tato data jsou přímo zpracovávána pouze pomocí kontrolní jednotky a přidružených obvodů pro práci s instrukcemi. K tomuto rozhodnutí mě vedlo, že není žádný důvod, proč by samotné instrukce měly možnost být zpracovány jinými obvody. Dalším důvodem je také rychlost. Pokud by veškeré instrukce procházely skrze datové sběrnice, tak by to zapříčinilo mnohem delší zpracování jednotlivých instrukcí. Každopádně je dobré se zmínit, že samotná data lze použít jako instrukci pouze pomocí časovače, který má v sobě uloženou adresu instrukce, na kterou má program přejít po uplynutí nastaveného času. To je způsobeno tím, že tato adresa je jako jediná přenášena pomocí datových sběrnic přímo do čítače.

Celé základní blokové schéma je vyobrazeno na obrázku 6.1. Toto zapojení má samozřejmě určité výhody i nevýhody. Hlavní výhodou je přístup k datům. Jelikož obě hlavní sběrnice A a B jsou tvořeny jako jednoduché registry, tak je možné data zpracovat zároveň pomocí ALU a zároveň je nastavit na výstupní brány. To velmi zkracuje čas, který by jinak byl potřeba při ukládání do registrů. Další výhodou je, že mohou být data zpracována například jako šestnáctibitová. Toho využívá hlavně Floating Point Unit (FPU), která je samotná postavena na šestnácti bitech. Více o FPU je napsáno v kapitole **6.6.**

Jednou z hlavních nevýhod této architektury je pomalejší výpočetní schopnost tohoto MCU. Důvodem je, že pokud chceme provést více operací skrze ALU, které na sebe navazují, například data sečíst a následně vynásobit, je nutné mezi každou operací data opět uložit do sběrnic a následně je znovu načíst. Proto tento MCU není vhodný pro složité aritmetické operace, jako je například konvoluce.

V případě potřeby, by bylo možné přidat k ALU zpětný registr, který by po každém výpočtu umožnil výsledné hodnoty vracet zpět do ALU. Následně by stačilo vložit dvojici MUX na vstupy do ALU. Tak by se dala přidat možnost pracovat s předchozími daty.

6.1 FUNKCE BLOKOVÉHO SCHÉMA

Celková funkce tohoto blokového schématu je vcelku jednoduchá. V prvním kroku řídící jednotka vyšle signál programovému čítači. Programový čítač inkrementuje svou hodnotu a pošle adresu dané instrukce do programové paměti. Programová paměť tudíž načte instrukci, kterou předá zpět řídící jednotce. V této jednotce je instrukce zpracována v několika hodinových taktech a provedena. Pokud instrukce obsahuje načtení dat z programu, tak skrze sběrnici jsou data uložena do A nebo B. Následně se kroky opakují. V případě, že je v instrukci provedení nějaké operace pomocí ALU, tak řídící jednotka vyšle signál, že má provést operaci ze vstupních dat. V té chvíli je výsledek stále na výstupu z ALU, ale není přiřazen k žádné sběrnici. K tomu dojde až ve chvíli, kdy řídící jednotka dostane instrukci „načti data z ALU a ulož do sběrnice A nebo B“.

Pro opakující se operace je možné vytvořit jednoduché podprogramy, což je zajištěno pomocí instrukce skoku. Pokud tedy řídící jednotka dostane signál o skoku, nejprve vyšle signál do zásobníku, aby uložil aktuální adresu instrukce. Po uložení této adresy vyšle signál do programového čítače, aby načetl novou adresu z programové paměti. Tím byl proveden skok. Po ukončení podprogramu následuje instrukce načtení adresy ze zásobníku. To probíhá pouze načtením adresy do programového čítače ze zásobníku. Zásobník se tím opět vymaže.

Další variantou, ke které může dojít, jsou samotné podmínky. Je jasné, že bez podmínek by se většina programů téměř nedala vytvořit. Ke kontrole podmínky slouží aritmetická jednotka. Podmínka se provádí pomocí podmíněného skoku. Pokud tedy řídící jednotka dostane instrukci k podmíněnému skoku, tak první zkontroluje, skrze signalizaci z ALU, jestli je podmínka splněna. Pokud ano, následuje podobný proces jako při skoku do podprogramu. Jedinou výjimkou je případ, kdy se nepožaduje návrat. V tu chvíli se adresa neukládá do zásobníku. V případě, že podmínka není splněna, řídící jednotka instrukci vynechá a načte rovnou novou instrukci.

Poslední variantou, ke které může dojít, je signalizace flag z časovače. Pokud časovač napočítá do předem určené hodnoty, tak je vyslán do řídící jednotky signál flag. Vzhledem k časové přesnosti, kterou musí časovač zajistit, má tento signál prioritu před ostatními. V tu chvíli tedy řídící jednotka dokončí právě prováděnou činnost a uloží aktuální adresu do zásobníku. Následně požádá o načtení adresy instrukce přímo z časovače. Po této operaci vynuluje čítač a zruší signalizaci flag. Po provedení těchto operací dále pracuje z načtené adresy. Načtená adresa by měla vždy směřovat do podprogramu, jinak by již nebylo možné odstranit původní uloženou adresu v zásobníku.

Veškeré tyto operace zajišťují provedení téměř jakéhokoliv programu bez nutnosti přidání dalších modulů. Jedinou výjimkou, kterou tento MCU nedokáže provést, je vnější přerušení. Tuto možnost lze ale kvůli zmíněné modularitě jednoduše přidat.

6.2 ZPRACOVÁNÍ INSTRUKCÍ

Dalším bodem, na který jsem navázal hned po vytvoření blokového schématu, bylo samotné zpracování instrukcí a s tím i samotná struktura daných instrukcí. Jistá komplikace, na kterou jsem narazil hned ze začátku, byla adresace. Při instrukci o délce osmi bitů je velmi jednoduché vyčerpat skoro celou instrukční sadu na adresaci registrů, toku dat a paměťových adres. Proto jsem se rozhodl pro variantu prodloužitelných instrukcí. To ve výsledku znamená, že například pro adresaci paměti RAM o adresové šířce šestnácti bitů bude použita z datového setu jen jedna instrukce pro ukládání a jedna pro načtení. Následná adresa bude uložena do dalších dvou bajtů instrukce, které hned na základní instrukci navazují.

Při potřebě budou mít tedy některé z daných instrukcí velikost až 24 bitů. Pokud by to ovšem platilo pro veškeré instrukce, tak by se zbytečně plýtvala programová paměť na téměř prázdné instrukce. Proto tento MCU vykonává některé instrukce po osmi bitech, některé po šestnácti a některé až po čtyřiadvaceti bitech. Tím je zajištěno, že nebude zbytečně plýtvána ani programová paměť, ani datová sada. Jedna z dvoubajtových instrukcí je ovládání datových sběrnic A a B. K tomu jsem přistoupil z důvodu, že je potřeba nastavit adresu, ze které se sběrnice nastaví. Proto každá z těchto sběrnic má samostatnou hlavní instrukci, která má v druhém bajtu především instrukci o adrese. Jedním ze zástupců instrukcí, které zabírají tři bajty, jsou instrukce skoku. Pokud bych se snažil tyto instrukce vytvořit jen pomocí jednoho bajtu, tak by byly téměř nerealizovatelné. To je z důvodu, že pro samotný skok existuje dvanáct různých variant, jelikož je potřeba určitý skok co nejpřesněji definovat. Další dva bajty v instrukci pak obsahují adresu, kam má být daný skok proveden, to ovšem neplatí pro skoky, které načítají adresu ze zásobníku.

Vzhledem k tomuto kroku vznikl prostor pro následnou úpravu, proto není instrukční set zaplněn celý. Vzhledem k této skutečnosti je tedy možné přidat minimálně dalších 100 instrukcí, které mohou ovládat další periferie.

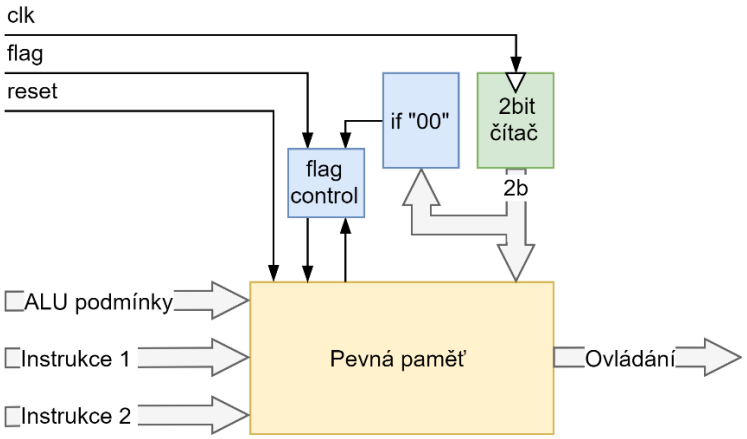
Většina vícebajtových instrukcí jsou, jak již bylo zmíněno, především instrukce s určitou adresací. Jedná se tedy o instrukce, které ovládají například paměť RAM. V této instrukci jsou tedy následující dva bajty určeny pro adresu v paměti. Další vícebajtová instrukce ovládá vstupní a výstupní porty. Tato instrukce má v prvním bajtu jen informaci, že se jedná o instrukci pro I/O. Následující jeden bajt pak adresuje bránu a ovládá její funkce.

Problém pak nastává při samotném zpracování instrukce. Původní myšlenka byla ta, že programový čítač nebude muset být nijak složitě řízen. To by ovšem při využití variabilní délky instrukce nešlo. Proto je zapotřebí v každém kroku nastavit programový čítač tak, aby inkrementoval svou hodnotu od jedné do tří, v závislosti na délce instrukce. O to se stará samotná řídící jednotka.

Tento problém se týkal i samotné programové paměti. Následovaly tedy dvě možnosti. Buď budou části této instrukce prováděny sekvenčně, což by znamenalo, že délka každé instrukce bude jinak dlouhá, anebo vytvořit možnost, kdy bude tato instrukce zpracována vždy najednou. Nakonec jsem tedy vyvedl z programové paměti vždy tři instrukce najednou. To sice zkomplikovalo samotnou programovou paměť, ale velmi to zjednodušuje a zrychluje chod samotného programu. Paralelní zpracování dlouhé instrukce tak umožňuje zkrátit čas na třetinu.

6.3 ŘÍDÍCÍ JEDNOTKA

Jedním z velmi důležitých bloků při zpracování instrukcí je řídící jednotka z angličtiny CU. Tato jednotka se stará právě o zpracování jednotlivých instrukcí a jejich interpretaci. CU má tedy na starosti ovládání všech bloků v celém MCU. Jelikož na provedení některých instrukcí je potřeba více kroků, tak i CU potřebuje pracovat ve více cyklech. Celá tato jednotka by se dala představit jako pevná paměť, kde vstupní adresou jsou vstupní instrukce, signály přerušení, a malý čítač pro oddělení více cyklů. Výstupy této „paměti" jsou pak následné ovládací signály pro jednotlivé bloky nacházející se v MCU.



Obrázek 6.2 Blokové schéma ovládací jednotky

Jak je vidět na obrázku 6.2, tak tato ovládací jednotka má k sobě přidružený dvoubitový čítač, který zajišťuje, aby každá z instrukcí byla provedena ve čtyřech krocích. Pro některé instrukce je to sice zbytečné a daly by se provádět mnohem rychleji. Na druhou stranu, pokud rozložíme i jednodušší instrukci do více kroků, zabráníme tak možným kolizím a hazardům. Jako příklad můžeme uvést ukládání do paměti. Tato instrukce by se dala provést v jednom hodinovém taktu. Může tak ale dojít k uložení špatných dat nebo do nesprávné adresy při zpoždění některého z hradel. Pokud ovšem v prvním kroku instrukce nejprve nastavíme požadovanou adresu a data, a až v druhém kroku se zapíšou hodnoty, tak k takové situaci nemůže dojít. Další výhodou provádění všech instrukcí ve stejném počtu hodinových taktů je jistá předvídatelnost časové náročnosti. Tím je myšleno, že pokud znám hodinový takt a počet instrukcí v jednotlivých podprogramech, jsem velmi jednoduše schopen říct, kolik času jednotlivé podprogramy zaberou.

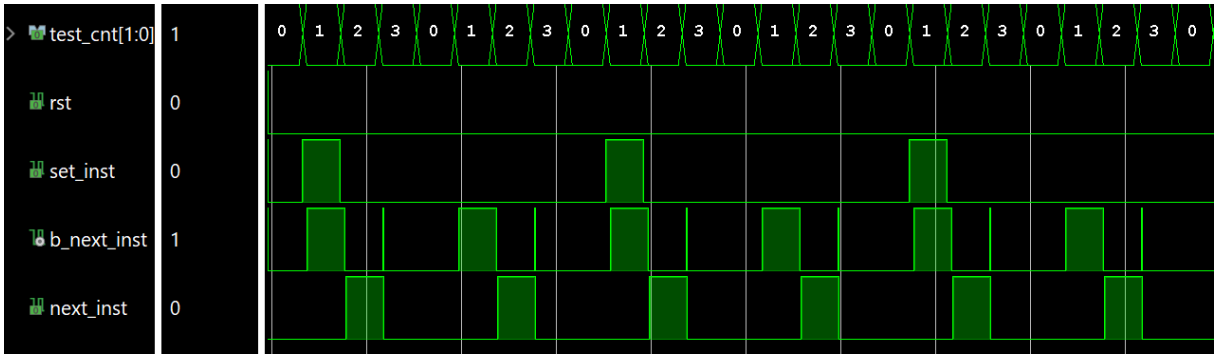
V úvodu této kapitoly se zmiňuji, že maximální délka jedné instrukce odpovídá délce 24 bit, tedy tří bajtů. Pro někoho tedy může přijít zvláštní, že podle obrázku 6.2, do řídící jednotky vstupují pouze dvě instrukční sběrnice. To je z důvodu, že třetí datová sběrnice není důležitá pro tuto jednotku. Jak již bylo zmíněno, poslední bajt je vždy

využíván jen pro adresaci dat. To ovšem není důležité pro řídící jednotku, ale přímo pro dané paměti, které tuto hodnotu potřebují znát.

I přes to, že jednotka nevyužívá poslední bajt, navíc k těmto dvěma bajtům je potřeba připočítat další hodnoty, které mění ovládací signály. Jako další do jednotky vstupují signály, které přichází z ALU. Je to přesněji kontrola podmínek. Tato alu kontroluje čtyři podmínky, z toho vyplývá, že tyto podmínky mají další dva vstupní bity.

Dalším vstupem je ovládací signál pro přerušení. Jelikož je ale každá instrukce prováděna ve více krocích, je potřeba, aby nedošlo k přerušení v rámci jedné instrukce. To je zajištěno pomocí ovládacího obvodu pro flag. Pokud tedy dojde k přerušení, tak jednotka dokončí poslední operaci, kterou provádí, a následně teprve provede vše potřebné v rámci sekvence pro flag. Tato jednotka musí být zároveň také ovládána, jelikož po dokončení flag, je potřeba ji vynulovat.

Dohromady by se tedy v překladu jednalo o „paměť", která má 22b adresu. Prakticky je ale jednodušší a úspornější tuto jednotku tvořit pomocí Look Up tabulek pro logické funkce než pomocí paměti. To je z důvodu, že některé z instrukcí v sobě přímo obsahují adresy, které by zbytečně plýtvaly touto pamětí. Pomocí logických funkcí se tak tyto adresy dají přímo propojit na výstupy a zjednoduší se tak zapojení. Proto je CU napsána pomocí podmínek ve VHDL, které jsou především **When Case** a **If else**. Při překladu tak dojde k vytvoření větších celků s multiplexory a jednoduchou logikou.



Obrázek 6.3 Ukázka sinchroniace signál next__inst

Při použití takovéto ovládací jednotky ovšem může docházet k nežádoucím jevům, které jsou hazardy. Jelikož každý ze signálů má svou vlastní kombinační logiku, tak je jasné, že vzhledem k zpožděním na hradlech může docházet k celkem velkým logickým hazardům anebo zpožděním mezi jednotlivými signály. Pro některá data by to téměř nevadilo. Příkladem jsou již zmíněné adresy, které jsou většinou použity až ve chvíli, kdy přijde nějaký aktivační signál. Větší problém může nastat při hazardech u ovládacích signálů, které by již mohly způsobit nesprávnou funkci celého mikrokontroléru. Aby nedocházelo k takovým situacím, tak tyto hlavní ovládací signály mají na výstupu registry, které jsou ovládány hodinovým taktem. To sice ve výsledku

zabrání jistým problémům s nestabilitou, ale naopak to sebere jeden hodinový takt ze čtyř pro každou instrukci. To je způsobeno tím, že každá část instrukce je provedena až na začátku následujícího taktu. Toto zpoždění a eliminace hazardů jsou vidět na obrázku 6.3, který pochází z implementační simulace. Na obrázku je vidět, jak signál **next_inst** je zpožděn oproti signálu **b_next_inst** a je eliminován logický hazard.

I přesto, že některé instrukce tento registr na výstupu nepotřebují, je dobré, aby byl na výstupu téměř pro každý signál. Důvodem je zbytečné překlápění hradel při stabilizaci dat. Musíme si uvědomit, že každé překlopení jednoho hradla může způsobit překlopení dalších stovek hradel, která jsou na sebe napojena. To by při pár hradlech nevadilo, ale při tak velkém množství může u FPGA začít docházet ke zbytečnému dynamickému přehřívání. Na tento problém jsem narazil, když byl skoro celý mikrokontroler hotový. Pro odstranění jsem tedy zvolil větší synchronizaci výstupů z kontrolní jednotky, která velmi snížila jak energetickou náročnost, tak teplotní gradient. Před provedením tohoto kroku mi ve shrnující zprávě o projektu vyšel výsledek o vysokém teplotním nárůstu, především v dynamické části.

Celá jednotka je napsána tak, aby se v ní dalo co nejjednodušeji vyznat. Proto je ke každé podmínce přidružen malý komentář, který označuje, jakou instrukci provádí. Je mi ovšem jasné, že pokud tuto jednotku bude zkoumat někdo jiný, tak bude mít velký problém se v ní vyznat. Proto zde píši pár základních bodů, jak je celá tato jednotka zapojena.

Jednotka je napsána pomocí procesu **main**, který má v citlivostním seznamu pouze hodinový signál. V první části je hned napsán dvoubitový čítač, který zajišťuje chod ve čtyřech krocích. Hlavními signály z tohoto čítače jsou **cnt**. Dále jsou vidět veškeré signály, které mají buffer. Každý tento signál je odlišen od ostatních pomocí předpony **b_**. Dále následuje samotné zpracování instrukcí. Nejprve jsou všechny instrukce děleny pomocí právě zmíněného vektoru **cnt**. První část má na začátku reakci na přerušení pomocí signálu **flag**. Ta nastaví při přerušení signál **b_flag_timers**, který ovládá ostatní tři části instrukce. To zajišťuje, že se přerušení provede až po skončení instrukce. Dále v každé části následují samotné instrukce, které jsou v podmínkách rozděleny vždy po čtyřech bitech. To je pouze pro kratší zápis. Takto jsou pak popsány veškeré kroky každé instrukce. V posledním kroku **cnt="11"** je především vracení všech signálů do defaultní hodnoty. Bližší popis jednotky je pak vložen do samotného VHDL.

6.4 PROGRAMOVÁ PAMĚŤ

V další části jsem se zaměřil na paměť pro ukládání programu. Hned na začátku jsem si uvědomil, že osmibitové adresování nebude stačit ani zdaleka. V tom případě by samotná paměť mohla obsahovat jen program o délce 256 instrukcí. V tom jsou samozřejmě započítány jen instrukce, které jsou tvořeny jedním bajtem. Proto jsem musel přistoupit k větší délce adresace o velikosti 16 bitů. Takto velká adresace už umožní uložit až$2^{16} = 65536$ programových instrukcí. Tento mikrokontroler bude tedy ve výsledku mít k dispozici 524 kB prostoru.

Samotná konstrukce je pak vytvořena velmi jednoduše. Jedná se o paměťový blok napsaný pomocí **Array_Flash** o velikosti 0 až 65534. Tato paměť má pak na výstupu vytvořeny tři osmibitové výstupy.

Program je do paměti zapisován rovnou při tvoření mikrokontroleru. Dlouho jsem uvažoval nad možností přeprogramování při samotném chodu. Nakonec jsem se ale rozhodl, že bude jednodušší mít paměť tvořenou pouhým zápisem ve VHDL. Proto v tomto bloku je napsán společně s pamětí i tento program pomocí hexadecimálního zápisu. Vzhledem k tomu, že se jedná o MCU napsaný pro FPGA, není potřeba, aby se dala tato paměť měnit. Stejně by bylo potřeba napojit desku k počítači a program nahrát.

Jelikož je tato paměť vytvořena takovýmto způsobem, dochází při syntéze k tomu, že je tento modul přeložen jako kombinační obvod. To je ve výsledku velmi užitečné, jelikož pro malé programy tak nebude spotřebována žádná paměťová buňka, která by jinak byla potřeba. Na druhou stranu tak samotný program rozhoduje o tom, jak velké množství logiky bude využito. Tento problém se ale dá obejít a přinutit syntezátor, aby vytvořil pro tuto buňku paměť. Jelikož většina programů pro ukázku je malá, tak jsem nechal, aby je syntezátor přeložil tímto způsobem.

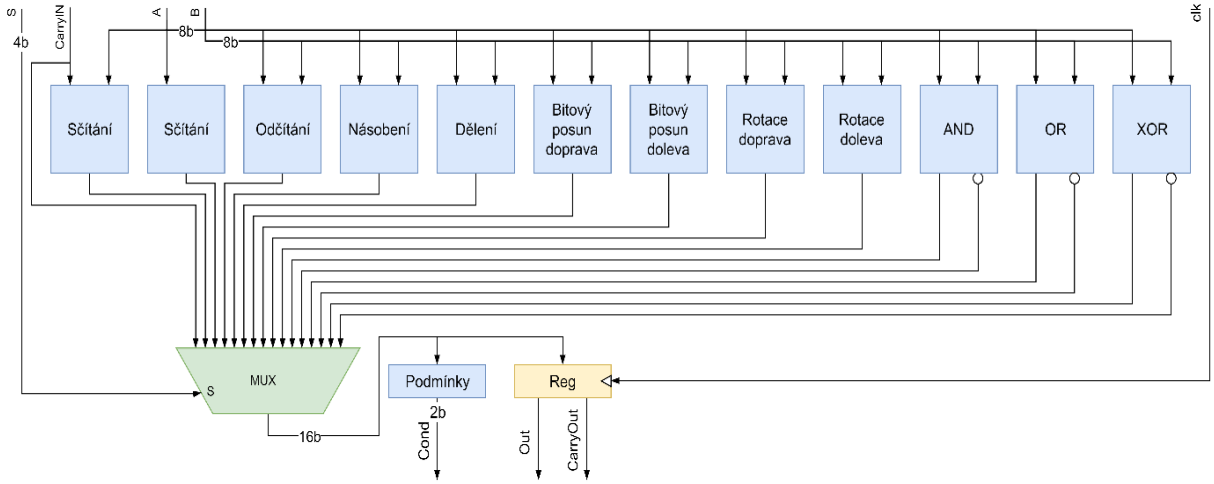
6.5 ALU

Jádrem všech digitálních zařízení jsou Aritmeticko-logické jednotky (ALU). Pomocí těchto jednotek probíhá většina jednoduchých operací. Mým cílem bylo vytvořit osmibitovou ALU, která umožní provádět veškeré základní operace v jedné instrukci. Tyto operace jsou:

- Sčítání a odčítání
- Násobení a dělení
- Bitový posun o N bitů doprava i doleva
- Bitová rotace o N bitů doprava i doleva
- Logické operace AND, OR, XOR, NOR, NAND a XNOR.

Pomocí těchto operací je tak možné provést jakoukoli složitější operaci. Je jasné, že některé z těchto operací se překrývají, což znamená, že pomocí dvou a více operací lze složit jinou operaci. To by ovšem neumožnilo vytvořit ALU, která by tuto operaci provedla v jedné instrukci. Tím by se ale nezlepšila výpočetní rychlost, která je zásadním cílem této ALU.

Zapojení ALU je tvořeno pomocí jednotlivých modulů pro provádění instrukcí, jejichž výstupy jsou sloučeny na výstupu ALU pomocí MUX. Taková konstrukce má především výhodu jednoduchosti zapojení. Nevýhodou je ovšem větší spotřeba energie, jelikož při změně dat jsou provedeny všechny operace najednou a až poté je vybrán požadovaný výstup. To ovšem není u ALU takový problém, jelikož je výstup následně synchronizován. Celé zapojení je pak vidět na obrázku 6.4.



Obrázek 6.4 Blokové schéma ALU

ALU má k sobě přidružen i menší registr, který se stará o přetečení. Pokud by tedy bylo potřeba zpracovat větší data, které ve výsledku budou mít větší velikost než osm bitů, je možné data přenést z tohoto registru zpět do ALU. Zapojení je vidět na obrázku 6.5.

Obrázek 6.5 Struktura ALU s registrem

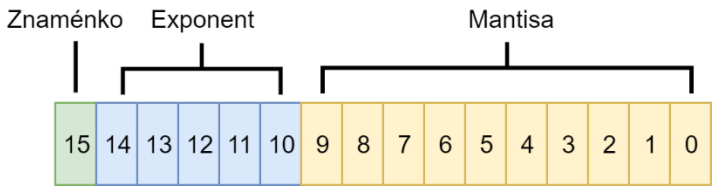
Pro ovládání tohoto vstupu do ALU slouží další dvě operace. Jedna z nich přenese na výstup z ALU pouze hodnotu, která je uložena v registru. Druhá operace tuto hodnotu přičte k hodnotě, která je na vstupu A.

Při použití takového zapojení je možné zároveň provádět operace i s většími vícebajtovými daty, jako například pro šestnáctibitová data. Operace s tak velkými daty ovšem nejsou již tak jednoduché a je potřeba zajistit více operací pro provedení jedné instrukce. To ovšem velmi usnadňuje právě přidružený registr.

6.6 FPU

Jednotka pracující s plovoucí desetinnou čárkou, Floating Point Unit (FPU), je již mnohem složitější. Jak bylo napsáno v kapitole věnované právě této jednotce, jedná se o blok, který pracuje s poněkud jinak zapsanými daty. Vzhledem k možnosti využití dvou hlavních sběrnic jsem se rozhodl implementovat šestnáctibitovou jednotku se základními čtyřmi aritmetickými operacemi: sčítání, odčítání, násobení a dělení. Pro zpracování dat tyto instrukce naprosto stačí. Problémem ale byla samotná reprezentace dat. Do FPU tedy nelze vložit klasická data ve tvaru unsigned jako do ALU. Bylo tedy potřeba vytvořit převodník, který data bude překládat z datového typu unsigned do typu float a zpět. Tento převodník je přidán jako další modul na datové sběrnice, lze ho tedy využívat libovolně bez aktivace FPU. To bylo hlavně z důvodu komplikovaného zapojení FPU. Při převádění dat do jiného formátu nebude MCU spotřebovávat zbytečně velké množství energie a tím se nebude zahřívat. S převedenými daty pak lze pracovat jako s jakýmikoliv jinými daty, což umožňuje i vyvedení těchto hodnot z mikrokontroleru ven. Vzhledem k využití šestnáctibitové FPU jsou i tato data tvořena dvěma bajty. To ovšem znamená, že pro provedení operace s těmito čísly je zapotřebí minimálně čtyř instrukčních kroků. Tato nevýhoda ovšem umožnila zpracovávat mnohem větší data. K tomuto kroku jsem se uchýlil hlavně z tohoto důvodu. Jak je vidět v kapitole 2.12, při použití FPU o velikosti pouhých osmi bitů by nemělo téměř smysl tuto jednotku implementovat, jelikož by to umožnilo pouze zpracovávat data o velmi krátkém desetinném zápisu.

Šestnáctibitový zápis formátu float se nazývá poloviční float (half float). Tento zápis je, jak již bylo zmíněno, jeden z menších zápisů, které jsou používány. Pro samotnou implementaci FPU je tedy dobré znát rozložení tohoto formátu. Tento formát tedy obsahuje: jeden bit pro znaménko, pět bitů pro exponent a deset bitů pro mantisu. Lépe je tento formát vyobrazen na obrázku 6.6 , kde čísla odpovídají očíslování jednotlivých bitů. Tento formát vychází ze standardu IEEE 754, který se právě stará o zápisy čísel.



Obrázek 6.6 Rozložení half float

Jak již bylo zmíněno, tento formát má mnohem lepší kapacitu, a proto do něj lze ukládat jak velmi malá desetinná čísla, tak i čísla s velkými exponenty. Dá se tedy říct, že při použití absolutní hodnoty má tento formát možnost uložit data od hodnoty $5{,}97 \cdot 10^{-8}$ až do největší možné hodnoty $65\,504$ Za největší hodnotu lze považovat nekonečno, s nímž by měla tato FPU pracovat. Nakonec jsem se rozhodl, že nekonečno jako takové vynechám z implementace, jelikož není potřeba ho využívat, a místo toho využiji zápis nekonečna pro další hodnoty čísel. Z toho plyne, že tato jednotka není úplně standardizovaná. Místo toho ovšem nabízí větší rozsah dat, a to až do velikosti 130 944. To, že jednotka není zcela standardizována, ovšem téměř nevadí, jelikož pokud by data byla vyvedena ven v tomto formátu, tak by se data větší než 65504 mohla prohlásit za Inf.

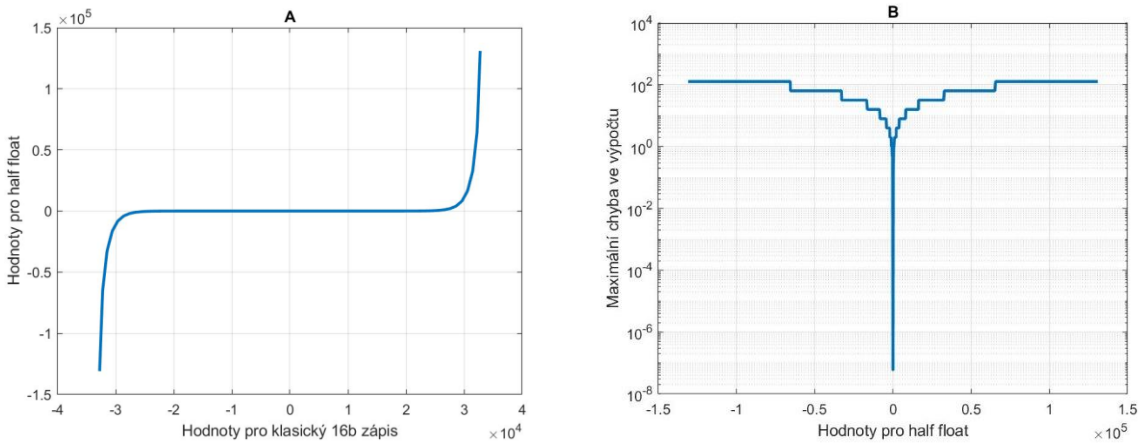
V tabulce 6.1 jsou vidět některé zajímavé hodnoty, které se dají vytvořit pomocí tohoto zápisu. Všechny hodnoty jsou ukázány jen v kladných číslech. Je ovšem jasné, že stejné hodnoty platí i pro záporná čísla, kde bude rozdíl jen v patnáctém bitu, který z těchto čísel udělá záporná čísla. Za zajímavost se dá považovat nula, která v tomto formátu může být jak záporná, tak kladná. Tato skutečnost se hodí při aplikaci dělení nulou, kdy je výsledek buď záporné nekonečno nebo kladné nekonečno.

Tabulka 6.1 Některé hodnoty zápisu half ploat pro mou FPU

	Binární zápis	Hodnota
nula	0 00000 0000000000	0
Nejmenší positivní číslo	0 00000 0000000001	0,000000059604645
Největší číslo menší než 1	0 01110 1111111111	0,99951172
jedna	0 01111 0000000000	1
Nejmenší číslo větší než 1	0 01111 0000000001	1,00097656
Největší možné číslo	0 11111 1111111111	130944

I když tento formát má oproti osmibitovému lepší rozsah dat a v tom případě i lepší přesnost, je jasné, že ani u tohoto formátu se nevyhneme jistým nepřesnostem. Příkladem je zlomek $\frac{1}{3} = 0,\overline{33}$. Tím, že toto číslo není možné zapsat jako součet čísel, která vyšla dělením dvěma, nemůžeme očekávat, že pro tuto hodnotu existuje přesný zápis, a proto je potřeba se k této hodnotě alespoň přiblížit. Nejbližší hodnota je tedy 0,33325195. Pro některé výpočty je tato přesnost dostačující. Každopádně, pokud

bychom od této hodnoty kupříkladu stále dokola odčítali hodnotu 10^{-5} , tak bychom měli jako výsledek stále stejnou hodnotu, která by se nikdy nezměnila. Proto je při výpočtech potřeba na tuto skutečnost dávat pozor.



Obrázek 6.7 (A) Rozsah FPU vzhledem k klasickému 16b unsigned. (B) Maximální chyba vzhledem k velikosti počítaných hodnot.

Jak je vidět na obrázku 6.7 (A), hodnoty jsou zapsány exponenciálně, což odpovídá exponenciální části zápisu. Je jasné, že čím větší hodnoty jsou v čísle uloženy, tím větší odchylku, musí mít. Na obrázku 6.7 (B) je vidět tato odchylka v závislosti na uložených číslech. Pro přehlednost je zvolen logaritmický graf, jelikož se chyba pohybuje přes několik řádů. Minimální chyba se tedy pohybuje okolo nuly, kde jsou uložena všechna malá desetinná čísla. Tato chyba je $\mu_{min} = 5{,}96 \cdot 10^{-8}$. Pokud se nepočítá chyba při přetečení, největší chyba nastává u největších hodnot a má velikost $\mu_{max} = 128$. Pokud bychom se zaměřili na relativní chyby, dojdeme k tomu, že veškeré výpočty mají stejnou relativní hodnotu, jelikož jejich chyba roste stejně rychle jako jejich hodnota. Veškeré výpočty tedy můžeme brát s přesností 0,1 %. Pokud by bylo potřeba zvýšit tuto přesnost, je možnost posunout nulovou hodnotu exponentu. Tím se ovšem zmenší maximální hodnoty, se kterými tato jednotka dokáže pracovat. Více je napsáno v kapitole 2.12.

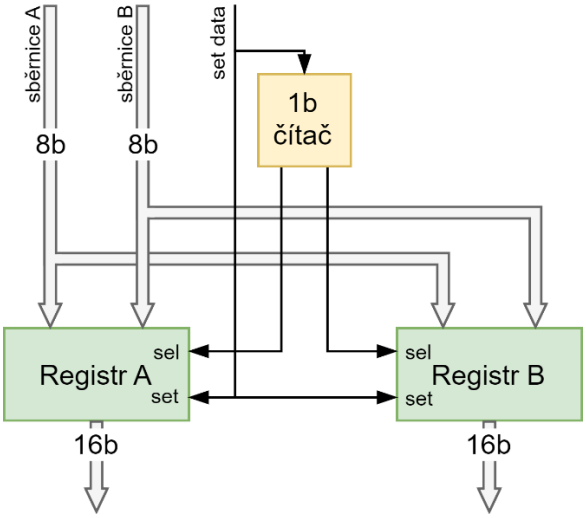
Protože je FPU velmi komplexní jednotka, která sama o sobě pracuje v několika krocích, je potřeba zajistit dostatek času na provedení jednotlivých operací. Jednotka bude potřebovat více času než čas pro provedení instrukce. To ovšem není tak zásadní problém, jelikož FPU dokončí operaci sama o sobě pouze s použitím hodinového signálu. Hlavním problémem je tedy čtení dat. Data nebude možné přečíst ihned v další instrukci, ale minimálně až v instrukci následující. Tento problém by se dal odstranit případnou signalizací nebo menší úpravou FPU. Bohužel by tak došlo ke větší složitosti zapojení. Proto stačí na tento fakt dbát a vynechat alespoň jednu instrukci po spuštění FPU.

6.6.1 IMPLEMENTACE FPU

V první části bylo potřeba promyslet blokové rozložení celého FPU. Jelikož je jednotka vytvořena pro šestnáctibitové výpočty, bylo nejprve potřeba vytvořit ukládání dat. Tato jednotka tedy pracuje ve třech krocích:

1) Načíst data A
2) Načíst data B
3) Provést operaci

Kvůli tomu je třeba provést minimálně tři operace, které toto ukládání postupně zajistí. Pro jednodušší ovládání se o toto stará jednoduchý čítač, který při vzestupném signálu na set_data ukládá střídavě mezi A_FPU a B_FPU. Po uložení těchto dat následuje provedení operace. Přehledně je toto načítání dat ukázáno na obrázku 6.8.

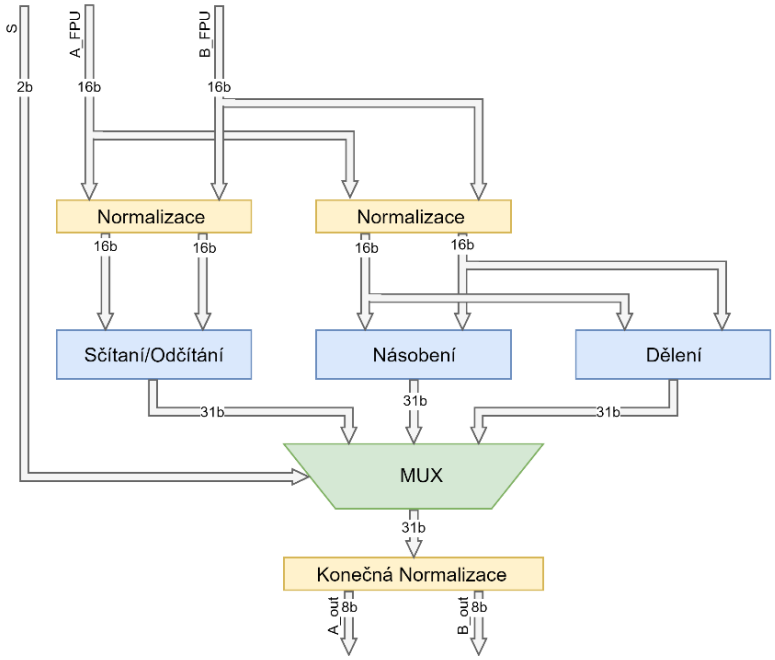


Obrázek 6.8 Struktura načtení dat do FPU

Po načtení dat lze provést požadovanou operaci. Vzhledem k tomu, že každá z operací je prováděna v několika krocích, bylo potřeba se zamyslet nad tím, jak tyto kroky implementovat do FPU. První myšlenka byla, že pro každý tento krok bude vyčleněn jeden modul, který tento krok provede. To je ovšem menší problém, jelikož některé z těchto kroků se mírně liší při různých operacích. Příkladem je normalizace dat. Při sčítání a odčítání je potřeba mít data trochu v jiném formátu než pro data násobení a dělení. Naopak poslední krok, který se zabývá normalizací na konci, může být stejný pro všechny operace, jelikož data vždy požadujeme ve formátu, v jakém mají být. Proto jsem zvolil takové zapojení, které bude co nejefektivněji provádět výpočty a nezabere tolik hradel na FPGA.

Výsledná implementace obsahuje dva normalizační bloky, jeden pro sčítání a odčítání a druhý pro násobení a dělení. Následně jsou tato normalizovaná data přivedena do bloku pro operace, který se skládá z více modulů. Tyto kroky probíhají vždy

pro všechna data. To je z důvodu, že jsou požadované výsledky separovány až následně po výpočtu. Po provedení všech operací jsou data vybrána pomocí MUX. Následně jsou přivedena výsledná data do bloku pro normalizaci, ve kterém se výsledek převádí zpět do formátu float. Po provedení normalizace jsou data vystupující ven přes datové sběrnice. Celková struktura výpočtů je ukázána na obrázku 6.9, kde je vidět i velikost jednotlivých datových sběrnic.



Obrázek 6.9 Bloková struktura zapojení FPU

Výstupy z veškerých bloků pro výpočet jsou vedeny pomocí 31b sběrnice.

To je z důvodu, aby nedošlo ke zkreslení výstupních dat. Větší velikost zahrnuje mantisu o velikosti 23 bitů a exponent o velikosti 7 bitů. Pro šestnáctibitový float má klasická mantisa velikost 10 bitů a exponent má 5 bitů. Mantisa byla zvětšena hlavně z důvodu, že při konečné normalizaci dochází k bitovému posunu. Dalším důvodem tohoto zvětšení je, že kromě samotných hodnot obsahuje také znaménko a dva bity před desetinnou čárkou mantisy. Pokud by nebyla data vypsána celá, mohlo by dojít k velkému zkreslení a výsledek by neodpovídal realitě. Stejně tak platí pro exponent, který může nabývat při násobení a dělení větších hodnot, než by se dalo uložit do klasického exponentu. Znaménkový bit zůstává stejný.

Je samozřejmě jasné, že taková data by nebyla reprodukovatelná. Proto jsou v konečné normalizaci data upravována opět do požadovaného tvaru. V tomto kroku dochází také k největšímu zkreslení, jelikož je potřeba zmenšit mantisu na 10 bitů. Tato chyba má maximální velikost podle Obrázek 6.7 (B).

6.6.2 PRVNÍ NORMALIZACE PRO FPU

Jak již bylo napsáno, normalizace před zpracováním má dvě varianty, jednu pro sčítání a odčítání a druhou pro násobení a dělení. Oba tyto bloky mají za úkol připravit data tak, aby se následná operace dala provést co nejjednodušeji. Oba bloky mají některé funkce společné. Jelikož FPU pracuje zvlášť s mantisou a znaménkem a zvlášť s exponentem, je potřeba tato data oddělit. Dalším krokem, který je pro oba bloky společný, je včlenění znaménka do mantisy. Mantisa má sice zápis podobný zápisu signed, ale přesto se jedná o úplně jiný zápis. Zápis mantisy je totiž vyjádřen klasickým součtem vah jednotlivých bitů. Následně znaménkový bit pouze udává, zda se jedná o kladné nebo záporné číslo. Oproti tomu je formát signed vyjádřen pomocí dvojkového doplňku. To v konečném důsledku znamená, že dvojkový doplněk vyjadřuje hodnotu, která je odečtena od zbytku čísla. Porovnání těchto zápisů je vidět v tabulce 6.2, kde poslední bit, označený červeně, vyjadřuje znaménko.

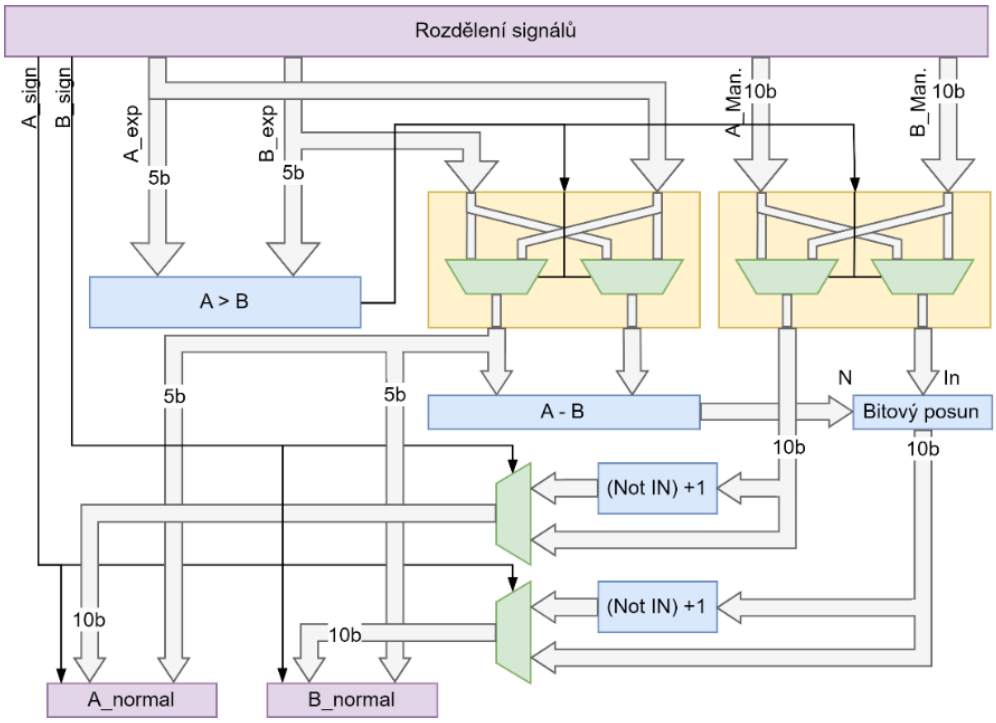
Tabulka 6.2 Porovnání rozdílů zápisů znaménkové mantisy a signed

	Zápis v osmibitovém formátu	hodnota
Zápis mantisy	10011100	-100
Zápis signed	11100100	-100

Při normalizaci pro sčítání a odčítání je navíc potřeba zajistit sjednocení exponentů a provést bitový posun mantisy podle rozdílu v exponentech. Aby nedošlo ke zkreslení, vždy se posouvá menší číslo tak, aby se exponent dorovnával směrem nahoru. Tím je tedy potřeba mantisu zmenšovat. Zmenšování je prováděno pomocí bitového posunu mantisy směrem doleva. U násobení a dělení není kromě zmíněného převádění potřeba žádná jiná operace.

Zapojení na obrázku 6.10 se může zdát celkově složité, ale nejedná se o nic komplikovaného. Jednotka, která je hned nahoře, jen rozdělí celou sběrnici na jednotlivé segmenty float. Dále v druhé části zjišťuje, jaký exponent má větší velikost, a podle velikosti se buď data přehodí, nebo zůstanou ve stejném pořadí. Následně se zjistí rozdíl, a o daný rozdíl se provede bitový posun u mantisy. Dále data pokračují do bloku, který je převádí na již zmíněný formát signed. To závisí na znaménku, proto jsou do těchto MUX přivedeny znaménkové bity. V poslední části se už jen sběrnice opět sloučí a pokračují k výpočtu.

Výpočetní jednotku, která se stará o samotné provedení operace, není třeba více popisovat. Jediná změna v této jednotce je při implementaci odčítání. Odčítání jako takové totiž není implementováno. Pro odčítání se v této jednotce, pro ušetření klopných obvodů, využívá právě formátu signed, pomocí kterého při odčítání změníme znaménko u druhého čísla a následně tyto hodnoty sečteme. Sčítání se provádí pouze pro mantisy s možností přetečení. Exponenty při sčítání a odčítání zůstávají stejné.



Obrázek 6.10 Blok Normalizace pro sčítání a odčítání

Normalizace pro násobení a dělení je mnohem jednodušší, protože stačí vyřešit jen znaménka. To opět provádíme pomocí převádění mantisy do typu signed. Tím je zajištěno, že při násobení nedojde k chybě ve znaménkách. Pro násobení je následná operace velmi jednoduchá, stačí sečíst exponenty, od kterých se ještě odečte nejvyšší bit, a následně se vynásobí mantisy mezi sebou.

6.6.3 OPERACE S ČÍSLY

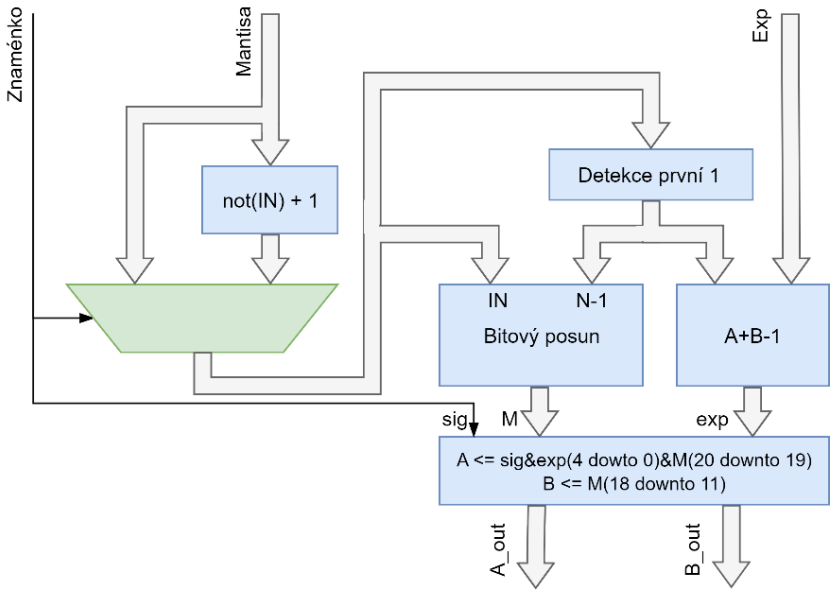
Pro operace je jasné, že se nedá využít jednoduchého zapojení pro ALU, jelikož se jedná o velmi odlišný datový zápis. Do všech bloků vstupují čísla rozdělená na mantisy **M1_sig** a **M2_sig**, které jsou definovány jako 12b signed, a exponenty **exp1** a **exp2**, které jsou ve formátu **std_logic_vector** a mají 5 bitů.

Při sčítání a odčítání je provedena jen operace mezi mantisami a výsledný exponent je stejný jako na vstupu operace. Pro násobení je výsledek vytvořen pomocí vynásobení mantis a sečtení exponentu. Od výsledku exponentu se musí ještě odečíst hodnota 14, aby nedošlo k zvětšení exponentu kvůli dodržení standardu IEEE 754.

Dělení bylo jednou ze složitějších operací, jelikož celý zápis mantisy pomocí signed není pro toto dělení úplně vhodný. Jde totiž o to, že veškerá čísla v mantise jsou brána jako desetinné číslo, které je ve tvaru **1,MMMM**, kde každé **M** značí jeden bit mantisy. Proto bylo potřeba celý dělenec před vydělením vynásobit hodnotou 1 048 576, která zajistí správnost výsledku. Hodnota exponentů je pouze odečtena od sebe, a přičítáme k ní hodnotu 16 ze stejného důvodu jako při násobení.

6.6.4 KONEČNÁ NORMALIZACE

Na konci je potřeba srovnat znovu mantisu do podoby zápisu **1,MMMM** a převést mantisu zpět z formátu signed. O tento převod se stará konečná normalizace. Jako vstupní data této jednotky je mantisa o velikosti 23b sběrnice a exponent o velikosti 7b sběrnice. Na začátku normalizace je nejprve převedena mantisa zpět pomocí oddělení znaménka, případné negace a přičtení hodnoty jedna.



Obrázek 6.11 Blokové schéma konečné normalizace

Dále následuje posun mantisy do normálového tvaru. Celý tento proces je prováděn pomocí detekce nejvyšší jedničky ve sběrnici mantisy. Cílem je zajistit, aby první jednička byla na místě 20 bitu sběrnice. Pokud to tak není, dochází k bitovému posunu a přičtení nebo odečtení hodnoty exponentu o počet těchto posunů. Následně už jsou jen data vyvedena ven. Celá jednotka normalizace je vidět na obrázku 6.11.

6.7 ČASOVAČ

V mikrokontrolerech se dá odměřovat přesný čas pomocí dvou způsobů. Jeden způsob je odměření času pomocí znalosti časové délky jedné instrukce. Pomocí toho pak lze provést **N** těchto instrukcí, a tak odměřit přesný čas. To má ovšem spoustu nevýhod. Hlavní nevýhodou je, že při čekání nemůže být provedena žádná jiná instrukce, nebo jen velmi málo instrukcí, které se dají provádět stále dokola. To ovšem znemožňuje dlouhé počítání času. Další nevýhoda se především týká mikrokontrolerů s proměnnou délkou instrukce. V těchto MCU je pak velmi složité spočítat přesnou délku odpočítaného času.

Druhou možností pro počítání času se používají integrované časovače, které dokážou odpočítat přesně stanovenou délku pomocí čítače hodinových signálů. Tyto časovače pak dokážou pracovat nezávisle na programu, jelikož uplynutí času signalizují

pomocí přerušení. Další velkou výhodou je, že takových časovačů může být integrováno víc, a tak lze počítat více časů najednou. Obsluha časovače je také velmi jednoduchá, stačí nadefinovat předřadný dělič frekvencí a počet cyklů. Následně už jen stačí pustit čítání. Po uplynutí časového okna čítač pozastaví pomocí přerušení program a provede se část podprogramu, která má být po tomto čase provedena. Následně stačí jen čítač opět restartovat.

V moderních mikrokontrolerech se časovače používají k více věcem. To je hlavně z důvodu, že při implementaci takovéto jednotky je výhodné, aby byla vytěžena co nejvíce. Dalším využitím časovače může tedy být generování PWM (pulzně šířková modulace). Pro přepnutí čítače do tohoto módu stačí nastavit děličku frekvence na délku jedné periody a následně nastavit čítač na požadovanou modulaci. Kromě PWM se pomocí časovače dá také měřit délka pulsu nebo s dalším modulem vytvořit jednoduchý integrační AD převodník.

6.7.1 IMPLEMENTACE ČASOVAČE

V případě tohoto mikrokontroleru je zvolena čtveřice jednoduchých osmibitových časovačů. Tyto časovače mají zabudovanou osmibitovou děličku, která dokáže vstupní frekvenci vydělit podle rovnice 6.1. Tento vydělený hodinový signál je dále veden do druhého čítače, který je také osmibitový. Ve výsledku je tedy možné, pomocí těchto čítačů, vytvořit periodu, která bude dlouhá 65536 period hodinového signálu, který vstupuje do časovačů.

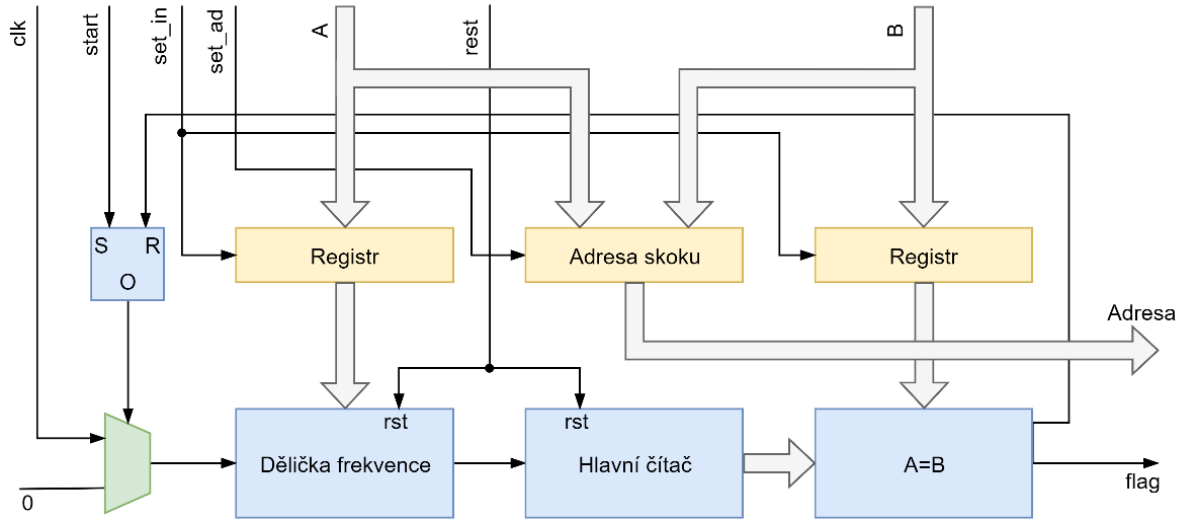
$$f_{Tin} = \frac{f}{256} \tag{6.1}$$

Obsluha čítače funguje v pěti základních krocích. V prvním kroku nastavujeme děličku frekvencí a hodnotu, do které má napočítat hlavní čítač. Tato data jsou přímo přivedena přes hlavní sběrnice A a B. Ve druhém kroku se nastavuje adresa, na kterou má program přejít, po dokončení čítání je opět tato adresa brána z A a B. Ve třetím se spouští čítání. Ve čtvrtém kroku čítač dopočítá do hodnoty, která byla předem stanovena a spustí signalizaci flag. V následujícím kroku musí řídící jednotka dokončit právě provádějící instrukci a následně resetovat čítač. Tím se také vynuluje flag.

Jelikož řídící jednotka nezareaguje ihned, dochází k menšímu zpoždění. Proto výsledný čas při velmi malých hodnotách nemusí být úplně přesný. Maximální chyba je tak dána časovou náročností instrukce, která je v tomto případě $4 \cdot T_{control}$ kde $T_{control}$ je délka jednoho taktu hodin vstupujících do řídící jednotky.

Jak je vidět na obrázku 6.12 , tak čítač v sobě obsahuje adresu instrukce, na kterou se má program přesunout po dokončení instrukce. Tento způsob jsem především zvolil z důvodu jednoduší orientace v tom, na co je použit daný čítač. Výhodou je také, že pokud

by bylo potřeba implementovat více čítačů, tak není potřeba vymýšlet nový prostor pro ukládání adres.



Obrázek 6.12 Bloková struktura čítače

6.7.2 PŘERUŠENÍ

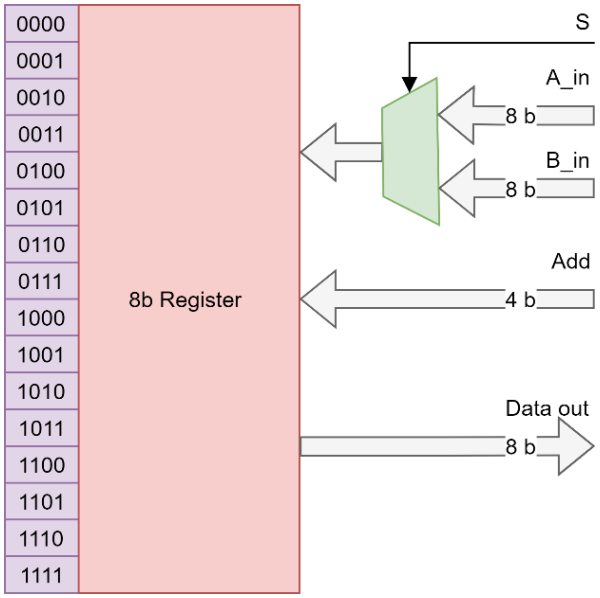
Jednou ze složitějších zapojení na tomto čítači bylo vytvořit ovládání příznaku přerušení. Pro to, aby přerušení vůbec fungovalo, bylo potřeba vytvořit ovládací elektroniku, která bude směřovat adresy přerušení a bude samotné přerušení zpracovávat. Nejprve bylo potřeba upravit řídící jednotku, která musí toto přerušení kontrolovat. Kontrola je vždy prováděna na začátku instrukce. Pokud tedy dojde k přerušení, nestane se, že by se vykonala pouze polovina instrukce. Při přerušení dostane řídící jednotka informaci, o jaký časovač se jedná. Pomocí této informace je pak možné časovač restartovat.

Jeden z nevyřešených problémů je priorita přerušení. Pokud se sejde více přerušení najednou, tak nastane jistý problém, jelikož jednotka je schopna vyřešit pouze jedno přerušení v jednom cyklu. Při takové situaci tedy dojde k tomu, že program se pokusí vykonat první přerušení. Přejde tedy na adresu, která je v tomto přerušení. Po skoku ale dostane řídící jednotka opět signalizaci o přerušení, proto přejde opět na jinou adresu. Následně je vykonána část podprogramu, který je pod druhým přerušením. Po dokončení této části, program přechází na předchozí adresu. Předchozí adresa je adresa prvního přerušení. Proto je provedeno i přerušení první. Z toho vyplývá, že přerušení, které přijde jako první, je provedeno jako poslední.

Tento problém by se dal vyřešit jednotkou, která by řešila správu jednotlivých přerušení. Tím by se dalo vyřešit i externí přerušení nebo přerušení od jiných komponent. Pro tento MCU jsem se nakonec rozhodl tuto jednotku neimplementovat, jelikož mi to prozatím nepřipadá důležité. Na druhou stranu je to jedna z možností rozšíření.

6.8 RAM A REGISTRY

Jednou z velmi důležitých jednotek pro běh programu je jednotka, do které se dají ukládat data. Hlavními požadavky pro tyto paměti jsou jejich jednoduchá přístupnost a hlavně rychlost. Dalším požadavkem je také velikost paměti. Zásadním problémem, na který jsem při tvorbě paměti narazil, bylo vyvážení mezi velikostí paměti a rychlostí přístupu k datům. Z tohoto důvodu jsou do MCU implementovány dva druhy pamětí. Tyto paměti jsou uživatelské registry a paměť typu RAM.



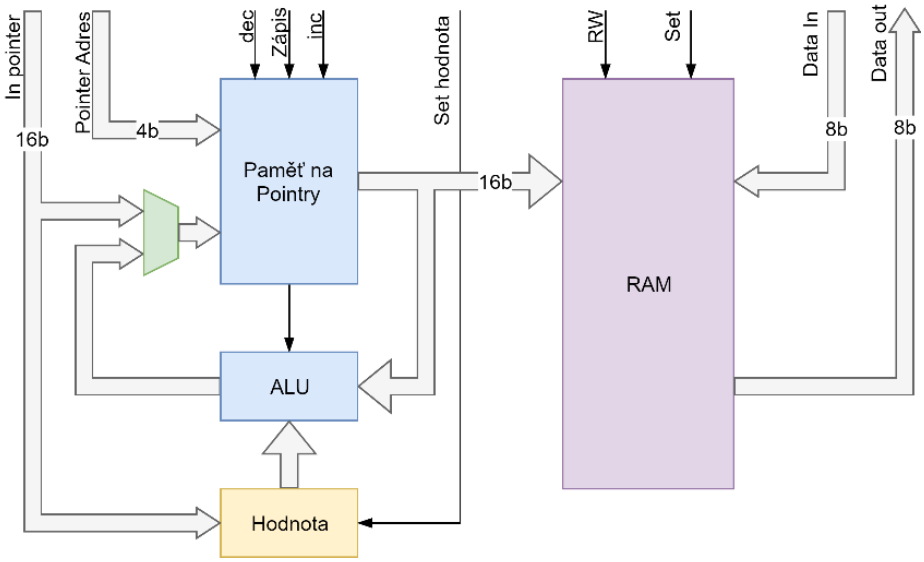
Obrázek 6.13 Struktura uživatelských registrů

Uživatelské registry jsou určené především pro rychlý přístup k datům. Jedná se totiž pouze o osmibitovou paměť s adresovou sběrnicí o velikosti čtyř bitů. Tato paměť tedy dokáže uložit pouze 16 B dat. Tato malá paměť má ovšem velkou výhodu v rychlosti zápisu
a čtení, jelikož je možné obě tyto operace provést v rámci jedné instrukce. Využití této paměti je tedy především pro taková data, ke kterým je nutný velmi rychlý přístup, nebo jsou používána velmi často. Tato paměť tedy může například sloužit pro ukládání koeficientů, které budou velmi často používány. Rychlost této paměti spočívá především v přímé adresaci. To ve výsledku znamená, že ukládání a čtení z této paměti je přímo napsané v programu. Není tedy možné v této paměti vytvářet například pole dat nebo ji adresovat pomocí ukazatelů. Celá struktura uživatelských registrů je vidět na obrázku 6.13.

Paměť RAM je navržena tak, aby do ní bylo možné uložit co nejvíce dat. Jedná se opět o osmibitovou paměť, která má šestnáctibitovou adresu. Ve výsledku má tedy kapacitu 65 536 B. Další výhodou této paměti je její řízení adres pomocí datových sběrnic. Veškerá adresace tedy prochází skrze datové sběrnice A a B. Tím je umožněna nepřímá

adresace, která přidává mnoho dalších možností, jak s daty pracovat. Příkladem je rychlé načítání většího množství dat, které budou zpracovávány postupně. Lze tedy vytvořit proměnou pointer, která bude ukazovat na konkrétní část v paměťovém prostoru. Tento pointer můžeme postupně inkrementovat a vytvářet tak pole dat.

Naopak velkou nevýhodou této paměti je, že pro uložení nebo načtení dat by bylo potřeba vždy uložit požadovanou adresu do sběrnic A a B. V případě inkrementace pointeru je tato operace ještě složitější, jelikož musíme k pointeru skrze ALU přičíst hodnotu jedna. To ovšem při všech požadovaných operacích pro provedení přičtení zabere zbytečně velké množství programové paměti a tím také zabere i velké množství času.



Obrázek 6.14 Paměť RAM s ovládáním

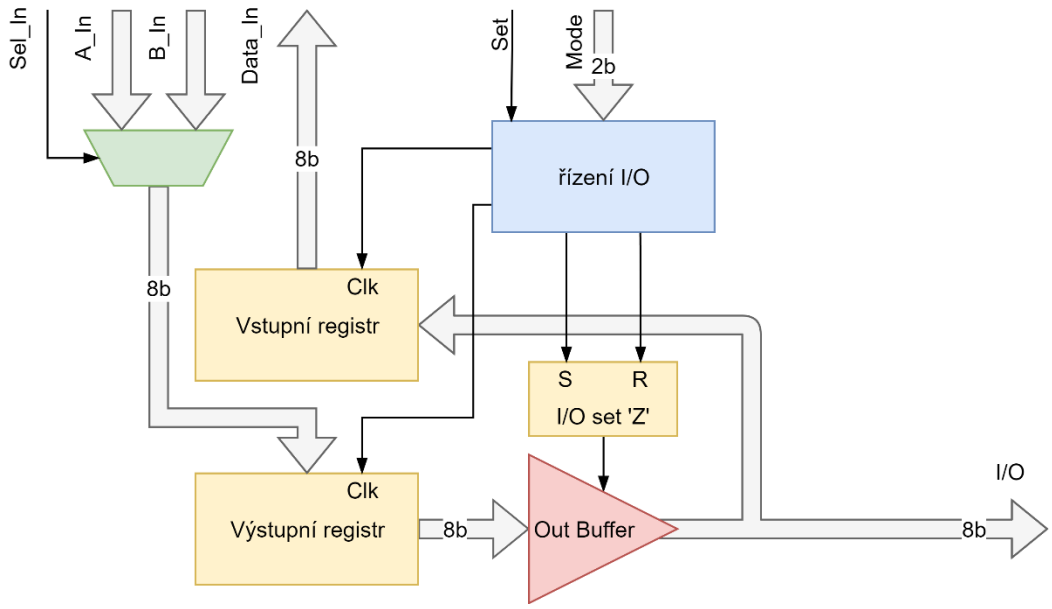
Pro vyřešení tohoto problému jsem společně s touto pamětí vytvořil modul, který se stará především o pointery nebo data, která se často inkrementují nebo dekrementují. Tato jednotka je přímo napojena na adresovou sběrnici paměti RAM, pomocí které velmi rychle nastavuje adresu pro čtení nebo zápis. Dále tato paměť má u sebe jednoduchou ALU jednotku, která pomocí jednoduchého příkazu dokáže k hodnotě pointeru přičíst nebo odečíst předem určenou hodnotu. Tato hodnota je v základu nebo při resetu rovna hodnotě jedna, jelikož nejčastější hodnota, která se k hodnotě pointeru přičítá nebo odečítá, je právě rovna jedné. Při potřebě je ale jednoduché tuto hodnotu přepsat na hodnotu, která je nastavena na sběrnicích. Jádrem tohoto bloku je paměťová jednotka, ve které budou veškeré pointery uloženy. Tato paměť má kapacitu pro šestnáct pointerů o velikosti šestnácti bitů.

Využití takového zapojení urychlí ukládání a načítání dat nejméně čtyřikrát. Jelikož jinak by bylo potřeba v každém cyklu tento pointer ukládat na sběrnice, připočítávat k nim hodnotu a znovu nastavovat jako adresu. Dále také ušetří část registrů, ve kterých

by tyto pointery musely být uloženy. Celé zapojení včetně připojení k paměti RAM je ukázáno na obrázku 6.14. Na levé straně je vidět datová sběrnice, která je přímo připojena na sběrnice A a B. Tato sběrnice slouží jako vstup základních hodnot pointerů. Dále je také vidět, že se pomocí této sběrnice nastavuje i hodnota pro ALU. Tím je umožněno, aby tato hodnota měla také velikost až 16 b. Je zřejmé, že téměř nikdy nebude potřeba inkrementovat pointer o celou velikost paměti. Ale je tím umožněno vytvářet větší celky dat, mezi kterými se dá přepínat pomocí větších hodnot.

6.9 BLOKY I/O

Nedílnou součástí každého mikrokontroleru jsou vstupně-výstupní brány. Je jasné, že samotné vnitřní funkce by se bez I/O obešly a veškeré vstupy a výstupy by bylo možné následně řešit přímo pomocí sběrnic A a B. Toto řešení by ovšem bylo velmi neefektivní a nepraktické. Problém by byl v tom, že by stejně bylo potřeba implementovat několik dalších bloků, které by fungovaly alespoň jako výstupní registry pro udržení hodnoty. Ještě větší problém by byl při řešení vstupu dat MCU. Tato data by se musela přivést přímo do MUX, který řeší vstup do sběrnic A a B. To by bylo nakonec velmi komplikované, jelikož by byla oddělena vstupní sběrnice a výstupní sběrnice.



Obrázek 6.15 Bloková struktura ovládání vstupně výstupních pinů.

Pro snadnější komunikaci s okolím je tedy vytvořena čtveřice vstupně-výstupních bank s označením A-D, kde každá z nich v sobě obsahuje osmibitovou vstupně-výstupní sběrnici. To dává dohromady až 32 nastavitelných vstupně-výstupních pinů. Jak již bylo napsáno dříve, tak každá z těchto bank se nastavuje celá najednou. Ve výsledku to znamená, že veškeré výstupní hodnoty jsou nastavované najednou po celém jednom bajtu. To samé platí pro nastavení vstupu a výstupu. Proto celá banka funguje jako vstup nebo výstup. Takové řešení je mnohem jednodušší pro ovládání a zapojení. Vzhledem

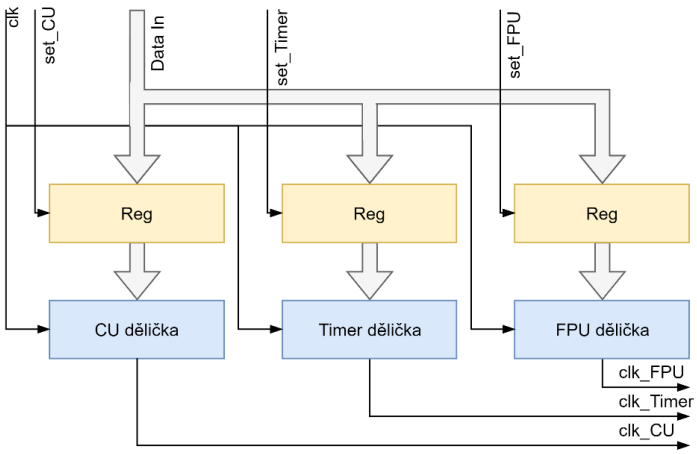
k tomu, že se jedná o MCU, který je implementovaný do FPGA, tak není ani potřeba vytvářet nastavení vstupu a výstupu na každý pin zvlášť.

Struktura vstupně-výstupních bloků je velmi jednoduchá, dá se říct, že se jedná o dvojici registrů, kde jeden slouží pro uložení výstupní hodnoty a druhý pro uložení vstupní hodnoty z datových sběrnic. Výstupní registry mají v sobě ještě zabudovaný buffer pro hodnotu vysoké impedance, na který se přepne v případě, že je celá banka přepnuta do režimu čtení. Tato hodnota je také nastavena jako výchozí, což slouží jako ochrana při spuštění. Důvodem je, že při restartu nebo zapínání MCU může být na některém z pinů nastavena vstupní hodnota logická 1. Pokud by ale tento pin byl nastaven jako výstupní s hodnotou logické nuly, došlo by ke zkratu a daný pin by se mohl zničit. Celé zapojení vstupně-výstupních bran je ukázáno na obrázku 6.15.

Výstup dat může být nastaven jak ze sběrnice A, tak ze sběrnice B, a to samé platí pro vstup dat. Jelikož je na vstupu registr, který zaznamená momentální hodnotu, tak je možné tuto hodnotu vyčíst až později, kdy bude potřeba. Tím se dá vytvořit jednoduché vzorkování přesných hodnot v momentu, kdy ještě nemohou být zpracovány.

6.10 MANAGEMENT HODIN

Veškeré bloky v celém MCU by samozřejmě nemohly fungovat bez hodinového signálu. Je jasné, že čím bude hodinový signál rychlejší, tím rychleji dokáže MCU zpracovávat jednotlivé instrukce. Ovšem ne vždy se hodí ta největší rychlost. Jde o to, že čím je větší rychlost, tím může častěji nastat situace, kdy se nestihne některý signál překlopit včas, a tím může celý MCU havarovat. Další problém, který přichází s vyšší rychlostí, je spotřeba a možné zahřívání. Spotřeba je hlavně problém u aplikací, kde chceme právě veškerou spotřebu minimalizovat. V takových případech se raději volí taková frekvence, při které bude MCU stíhat zpracovávat vše, co má, a zároveň nebude brát tolik energie.



Obrázek 6.16 Bloková struktura distribuce hodin

Další věcí je, že v MCU nepotřebují veškeré komponenty pracovat ve stejné rychlosti. Dobrým příkladem jsou časovače, u kterých na vstupní frekvenci závisí nejdelší doba, do které umí čítat. Představme si situaci, kdy potřebujeme provádět určité měření každou celou hodinu. Při použití tohoto MCU můžeme k tomuto účelu použít čítače z kapitoly 6.7 . Brzy ovšem dojdeme k závěru, že takový čas nepůjde změřit. Je to z toho důvodu, že bychom i při nastavení nejvyšších hodnot čítače požadovali vstupní frekvenci do těchto modulů o 18 Hz. To je nesrovnatelně menší frekvence oproti klasickým MCU, které běží řádově na jednotkách MHz.

Řešení takovýchto problémů spočívá ve využívání interního řízení hodinových signálů, které jsem se rozhodl implementovat i do tohoto MCU. V tomto případě se jedná o jednoduchou trojici nastavitelných osmibitových děliček frekvence. Jedna dělička se stará o hodinový signál pro řídící jednotku, druhá pro časovače a třetí pro FPU. Tyto jednotky mají klasické řízení pomocí programu. Jejich defaultní hodnota je nastavena na hodnotu vstupního hodinového signálu. Ve výsledku to tedy znamená, že po zapnutí MCU bude vše fungovat na této frekvenci, dokud děličku nezměníme. Celá struktura je vidět na obrázku 6.16 , kde je vidět, že se jedná o velmi jednoduché zapojení.

Minimální frekvenci lze na výstupu těchto děliček nastavit až na 1/256 původní frekvence. Ve výsledku to znamená, že pokud budeme jako vstupní frekvenci používat defaultní hodnotu 50 MHz, která byla využita při vývoji, tak můžeme používat frekvence od 195 kHz do 50 MHz. Tento rozptyl by měl být dostatečný. Pokud ovšem bude potřeba ještě nižší frekvence, zajistíme to pomocí snížení vstupní frekvence, která spodní hodnotu nemá omezenou.

6.11 INSTRUKČNÍ SADA

Pro využití veškerých bloků je potřeba mít správně vytvořenou instrukční sadu tak, aby využívala celý potenciál všech bloků. U klasických mikrokontrolerů jsou instrukční sady vytvářeny tak, aby také využily celý prostor délky všech instrukcí, což ve finále znamená, že pokud využíváme osmibitový MCU, tak bude mít ve většině případů využito celkově všech 256 instrukcí. Každopádně to také závisí na typu a využitelnosti.

V tomto případě jsem se snažil veškeré instrukce optimalizovat tak, aby nezabíraly zbytečně velké množství instrukčního prostoru. Proto je velké množství instrukcí rozloženo do více bajtů, jak bylo zmíněno v kapitole 6.3 o řídící jednotce.

Jednotlivé instrukce jsou brány i jako jednotlivé nastavení daných jednotek. Příkladem je již mnohokrát zmiňovaný registr, který pro svůj zápis a čtení zabírá dohromady až 64 různých instrukcí vzhledem k adresování. Pokud tímto způsobem spočítám počet všech potřebných instrukcí pro ovládání tohoto MCU, tak dojdu k výsledku, který je v tabulce 6.3.

Tabulka 6.3 Počet potřebných instrukcí pro ovládání jednotlivých Funkcí

Bloky nebo funkce	Počet potřebných instrukcí
restart	1
I/O A-D	32
Skoky	393222
Nastavení sběrnic A a B	524
ALU	16
Registry	64
Časovače A-D	16
FPU + převody	19
RAM + pointery	259
Nastavení hodin	3
Celkem	394 156

Je jasné, že tolik různých instrukcí opravdu nelze uložit do jednoho bajtu. Jak je vidět, tak 99 % veškerých instrukcí zabírají pouze skoky. To je způsobeno tím, že existuje 12 různých skoků. Šest z těchto skoků využívá adresu, která je přímo zapsána v programu. Protože v tomto MCU je využívána 16b adresa pro instrukce, tak pro přesné určení cíle skoku je potřeba 65 536 různých instrukcí. To při potřebě šesti takových instrukcí dává tuto hodnotu. Podobný případ je při nastavování sběrnic. Důvodem je, že jedna z instrukcí nastavuje přesnou hodnotu z ROM. Jelikož tato hodnota má 8b šířku, tak je potřeba na nastavení této hodnoty minimálně 256 různých instrukcí.

Je potřeba, aby některé z těchto instrukcí byly převedeny do více bajtové instrukce. Při takovémto zmenšení následně vznikne instrukční sada, která obsahuje 94 instrukcí, které mají různou délku. Důvodem pro tak razantní zmenšení je především již zmíněná možnost rozšiřitelnosti o další moduly. Dále byly tyto instrukce tvořeny především tak, aby byly co nejvíce uživatelsky přívětivé, a tím se dalo jednoduše poznat a pochopit, jak fungují jednotlivé bloky. Pro jednodušší pochopení a popis jsou instrukce roztříděny do 32 nezávislých instrukcí s různými parametry. Popis a vysvětlení veškerých instrukcí se nalézá v příloze C.

6.12 SHRNUTÍ PROJEKTU S PARAMETRY

Základem celého mikrokontroleru je jeho jednoduchost, kvůli které dokáže pracovat bez problémů i při frekvenci 100 MHz. To znamená, že při potřebě čtyř taktů na zpracování instrukce, dokáže vykonávat instrukce o frekvenci 25 MHz. Z této frekvence se dá vypočítat datová propustnost obou sběrnic A a B, která je pro každou 200 Mb/s. Pokud bychom uvažovali nad maximální rychlostí, kterou lze data ukládat na výstupní bránu, tak je možné se dostat až na 50 Mb/s. Toto zpomalení oproti rychlosti sběrnice spočívá v nutnosti načíst data, uložit je do sběrnice a následně data uložit do I/O. Jelikož je ale celá brána ovládána jako celek, tak to ve výsledku znamená, že výstupní piny lze obnovovat rychlostí 6,25 MHz. Taková rychlost je dostatečná pro ovládání mnoha komponentů jako jsou diody, datové sběrnice nebo třeba i displeje. Ve stejné rychlosti dokáže tento MCU i samplovat vstupní data. Je dobré zmínit, že veškeré tyto parametry byly měřeny pouze při defaultním nastavení jak strategie syntézy, tak implementace. Je možné, že při jiných strategiích by se nedalo dostat ke stejným hodnotám.

Co se týče výpočetní rychlosti ALU, tak pokud budeme uvažovat i nad přivedením dat do ALU a uložení dat, tak jsme schopni provést libovolnou operaci pomocí pěti instrukcí, což znamená, že jsme schopni provádět jednotlivé operace s frekvencí 5 MHz. Tato rychlost platí ale pouze pro ALU. Trochu jiný případ je FPU, u které je potřeba provést pouze šest operací na uložení dat do FPU. Dále je potřeba provést operaci, počkat alespoň dvě instrukce na provedení operace, a následně data opět uložit, což zabere čtyři instrukce. Ve výsledku to tedy znamená, že jsme schopni provést jeden šestnáctibitový výpočet v FPU o frekvenci 2 MHz. To znamená, že FPU má výkon 2 MFLOPS.

Dále tento mikrokontroler obsahuje 65 kB programovou paměť a 65 kB paměť RAM. K těmto pamětem je přidán rychlý registr, který má paměť pouze 16 B. Pro jednodušší ovládání paměti RAM je přidána paměť pro pointery, do které se vejde 16 pointerů o velikosti 16 b. Poslední implementovanou pamětí je šestnáctiúrovňový zásobník. To dává možnost vytvářet více podprogramů.

Dále mikrokontroler dokáže vykonávat jednoduché přerušení od čtveřice časovačů, které dokáží měřit časový úsek od teoretických 40 ns, kdy již má smysl tento časovač využít, až po 168 ms. Maximální hodnota je počítána při využití maximálního děliče do časovačů pomocí managementu hodinových signálů. Je jasné, že tato hodnota nestačí na delší časové úseky. Pokud by tedy bylo potřeba měřit například hodinové intervaly, tak je možné využít programového počítání uplynulých přerušení, nebo zpomalit vstupní hodiny.

Jelikož se jedná o mikrokontroler, který je implementován do FPGA, tak jsou také důležité parametry pro využití pole. Konkrétně pro toto pole se využití dělí do více kategorií, protože nás zajímá využití jednotlivých bloků, které jsou k dispozici.

Tabulka 6.4 Využití FPGA na vytvoření MCU

Bloky	Použitých	K dispozici	Využití [%]
LUT	1623	63400	2,56
LUTRAM	24	19000	0,13
Flip-Flop	546	126800	0,43
Block RAM	16	135	11,85
DSP	1	240	0,42

Celkové využití se ale může lehce lišit pro různé programy, jelikož jak již bylo zmíněno v kapitole 2.10, tak je program tvořen většinu času jako LUT tabulky než samotné uložené hodnoty v bloku paměti. Dá se ale říct, že tato změna není tak velká. Nejvíce by se lišila v případě velkého programu, kdy by již mohlo dojít k využití programového bloku. To by ovšem zabralo pouze o jeden blok paměti více než v tomto případě. Je také jasné, že se tyto hodnoty mohou velmi lišit při použití jiné strategie syntézy a implementace. Veškeré hodnoty v tabulce 6.4 platí pouze na defaultní nastavení programu Vivado.

7 UKÁZKOVÉ PROGRAMY

Co se týče hardwarového vybavení tohoto mikrokontroleru, bylo již v předchozích kapitolách napsáno mnoho. Z těchto informací si člověk dokáže asi přibližně představit, k čemu se dá tento MCU použít. Tyto informace se ovšem hodí lidem, kteří by se zajímali více o zapojení než samotné využití. Člověka, který se věnuje vývoji programů do MCU, by mohlo více zajímat, co vše se dá s tímto mikrokontrolerem naprogramovat a jak je jeho ovládání složité.

Hned v základu musím napsat, že k tomuto mikrokontroleru jsem nevyvinul žádný překladač. Proto není možné na tento MCU vyvíjet programy například pomocí jazyka C, Python ani pomocí Assembleru. Celé programování totiž probíhá pomocí zadávání hexadecimálního kódu samotných instrukcí do paměti v jazyce VHDL. Z toho je jasné, že programování tohoto MCU není vůbec jednoduché a je potřeba velká trpělivost. Dále z toho také plyne, že je téměř nemožné vytvořit nějaký složitější program, který zvládne ovládání velkého množství různých komponent. Jak již bylo ale psáno v několika kapitolách, tento MCU je více zaměřen jako výuková pomůcka než nějaký funkční blok, který by byl vhodný k využití. Na druhou stranu se toto vůbec nevylučuje.

Program je tedy potřeba psát v hexadecimálním nebo také binárním zápisu. Veškeré instrukce jsou k dispozici v příloze C. Další místo, kde jsem veškeré instrukce

zanechal, je přímo v bloku ROM, kde tyto instrukce s jednoduchým popisem naleznete v záhlaví modulu napsané v komentáři.

V této kapitole představím pár programů, které postupně ukazují funkce všech jednotlivých funkčních bloků. Pro jednodušší ukázku budu tyto programy testovat přímo v simulačním prostředí programu Vivado, aby bylo mnohem jednodušší ukázat jednotlivé výstupy a případně jednotlivé důležité signály. Pár jednodušších programů vložím pro ukázku přímo sem jako kód, zatímco ostatní programy, které jsou mnohem delší, budou přidány do přílohy. Dále jsou všechny tyto programy opět vloženy do modulu ROM. Tentokrát jsou k nalezení na konci modulu. Pokud by bylo potřeba tyto programy využít, je jednoduché je zkopírovat, vložit do prostoru pro program a zrušit jejich zakomentování. Na konci každého programu je nastavena pro všechny ostatní adresy hodnota 53, tato instrukce totiž neprovádí nic. Z toho důvodu je vhodná na ukončení programu.

7.1 TEST ALU

Pro jednoduché otestování ALU je nejlepší způsob zkusit vypočítat nějaký jednoduchý příklad s co nejvíce různými operacemi. Pro tento účel jsem si zvolil následující příklad:

$$\frac{(10 * 2) - 5}{3} = 5 \qquad 7.1$$

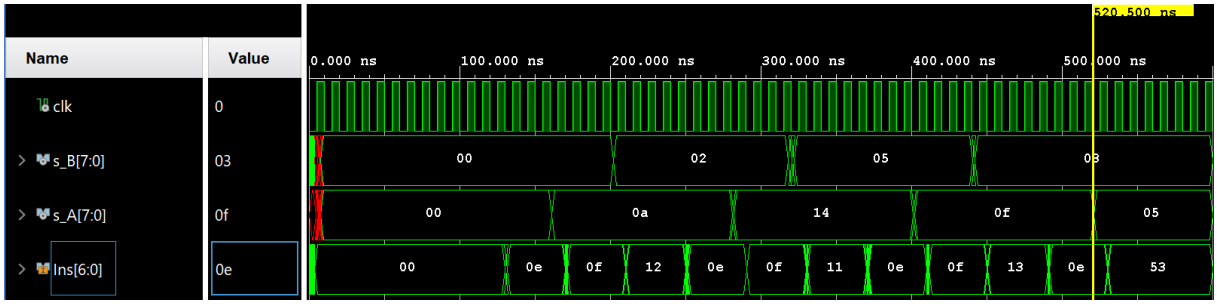
Pro takto jednoduchý příklad není ani potřeba využívat paměť, a veškeré výpočty mohou proběhnout pouze pomocí sběrnic A a B. Nejdříve je potřeba tento příklad rozložit na jednotlivé operace a poskládat je tak, aby výsledek byl správný. To znamená, že nejprve bude provedeno násobení, následně odčítání a nakonec dělení.

```
INST      ADD  FUNKCE      INST      ADD  FUNKCE
x"00", -- 01 - reset       x"10", -- 0C - from rom
x"0E", -- 02 - set A       x"05", -- 0D - value
x"10", -- 03 - from rom    x"11", -- 0E - ALU -
x"0a", -- 04 - value       x"0E", -- 0F - set A
x"0F", -- 05 - set B       x"20", -- 10 - from ALU
x"10", -- 06 - from rom    x"0F", -- 11 - set B
x"02", -- 07 - value       x"10", -- 12 - from rom
x"12", -- 08 - ALU *       x"03", -- 13 - value
x"0E", -- 09 - set A       x"13", -- 14 - ALU /
x"20", -- 0A - from ALU    x"0E", -- 15 - set A
x"0F", -- 0B - set B       x"20", -- 16 - from ALU
```

Obrázek 7.1 Program spravující ALU

Celý program je vidět na obrázku 7.1, kde jsou jednotlivé instrukce napsány vždy na začátku řádku. Ke každé instrukci jsem dále přidal její adresu a vysvětlení, co tato instrukce dělá. Jak je vidět, tento program pouze načítá hodnoty z paměti ROM a přidává je do výpočtu tohoto programu. Na konci tohoto programu bude výsledná hodnota zapsána na sběrnici A. Jednou ze zajímavostí je první instrukce, která říká všem modulům,

aby se resetovaly. To je pro případ, že by někde zůstaly hodnoty, které by mohly narušit celý chod programu.



Obrázek 7.2 Simulace programu v post implementační časové simulaci

Celý tento program zabere jen velmi málo času. Jak je vidět podle simulace, tak pro provedení takové operace je potřeba pouze 520 ns. Veškeré hodnoty, které jsou v simulaci, jsou zapsány v hexadecimálním zápisu. Proto když se podíváme na sběrnici A a B na obrázku 7.2, zapsané jako s__A a s__B, tak si můžeme všimnout na s__A veškerých mezivýpočtů. Na konci je pak vidět již hodnota 5, ke které se měl výpočet dostat.

7.2 REGISTRY

K otestování registrů opět použijeme jednoduchý výpočet, který ovšem nelze vykonat bez uložení některých výsledků do mezipaměti.

$$3 * 5 + 4 * 2 = 23 \qquad 7.2$$

Tento výpočet je opět velmi jednoduchý, na druhou stranu je jasné, že bez použití registru ho nelze vykonat. Proto bude potřeba nejprve vynásobit první dvě čísla, jejich výsledek uložit do registru, a následně teprve vynásobit další dvě čísla. Nakonec se hodnota opět načte z registru a přičte se k ní výsledek. Celý tento postup je vidět v programu na obrázku 7.3.

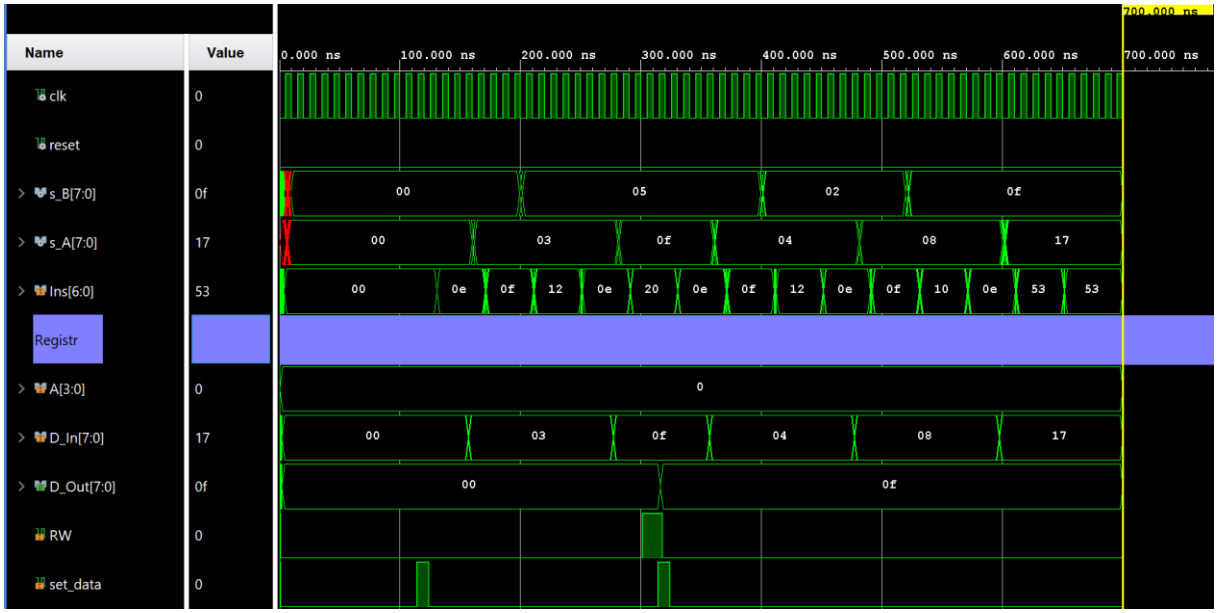
```
INST      ADD  FUNKCE
x"00", -- 01 - 0
x"0E", -- 02 - set A
x"10", -- 03 - from rom
x"03", -- 04 - value
x"0F", -- 05 - set B
x"10", -- 06 - from rom
x"05", -- 07 - value
x"12", -- 08 - ALU *
x"0E", -- 09 - set A
x"20", -- 0A - from ALU
x"20", -- 0B - save A to reg 0000
x"0E", -- 0C - set A
```

```
INST      ADD  FUNKCE
x"10", -- 0D - from rom
x"04", -- 0E - value
x"0F", -- 0F - set B
x"10", -- 10 - from rom
x"02", -- 11 - value
x"12", -- 12 - ALU *
x"0E", -- 13 - set A
x"20", -- 14 - from ALU
x"0F", -- 15 - set B
x"00", -- 16 - from reg 0000
x"10", -- 17 - ALU +
x"0E", -- 18 - set A
x"20", -- 19 - from ALU
```

Obrázek 7.3 Program pro test registrů.

Jak již bylo psáno v kapitole o registrech a pamětech, tak registry jsou vytvořeny tak, aby s nimi šlo pracovat co nejjednodušeji, a proto se dají využít v rámci jedné instrukce.

Tento program se může zdát velmi dlouhý na to, kolik operací je v tomto programu provedeno. Na druhou stranu si musíme uvědomit, že spousta z těchto instrukcí jsou vícebajtové. To znamená, že reálně bude provedeno v tomto programu dohromady pouze 13 instrukcí. Na konci tohoto programu by tedy na sběrnici A měla být hodnota 23, což odpovídá hexadecimálnímu zápisu čísla 0x17. Po simulaci vidíme tuto hodnotu na konci viz obrázek 7.4.



Obrázek 7.4 Simulace testování registrů

Pro zajímavost je v simulaci ukázán i samotný modul pro registry, na kterém je vidět, jak celý tento modul pracuje. Při instrukci 0x20 dojde k zapsání hodnoty do registru s adresou 0x00, která se okamžitě propíše na výstup registru, jelikož je stále nastavena stejná hodnota. Následně je vidět v čase 500 ns, že dojde opět k načtení této hodnoty. Doba trvání tohoto programu je do vypočítání výsledku dlouhá 600 ns, což je o něco málo více než v programu předchozím.

7.3 PODMÍNĚNÉ SKOKY A ZÁSOBNÍK

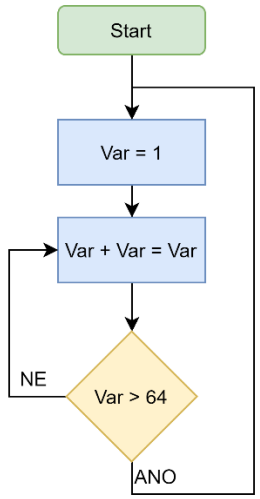
Vytváření různých programů se samozřejmě nedá obejít bez podmínek a skoků do podprogramů. Jak již bylo psáno dříve, podmíněné skoky jsou možné pouze pomocí tří základních podmínek, ze kterých se dají následně složit další podmínky, které by mohly být potřeba. Dalším modulem, který bude tímto programem otestován, je zásobník, do kterého se ukládá adresa místa, odkud program skočil jinam.

Program, který tyto funkce bude testovat, musí tedy obsahovat více než jen pár výpočtů. K tomu se bude hodit nějaký program, který bude fungovat cyklicky a bude vypisovat nějaké určité hodnoty. K tomu se velmi hodí například výpis hodnot geometrické posloupnosti. V této posloupnosti platí, že následující hodnota je dvojnásobkem hodnoty předchozí. Pro zjednodušení tohoto programu tedy vytvořím

podprogram, který vezme hodnotu, vynásobí ji dvěma a uloží zpět. Jelikož chceme ověřit i schopnost podmínek, tak vytvoříme podmínku, která bude hlídat, jestli je výsledná hodnota stále menší než hodnota 64. Ve chvíli, kdy tato hodnota nebude platit, tak začne celý program počítat znovu od hodnoty jedna.

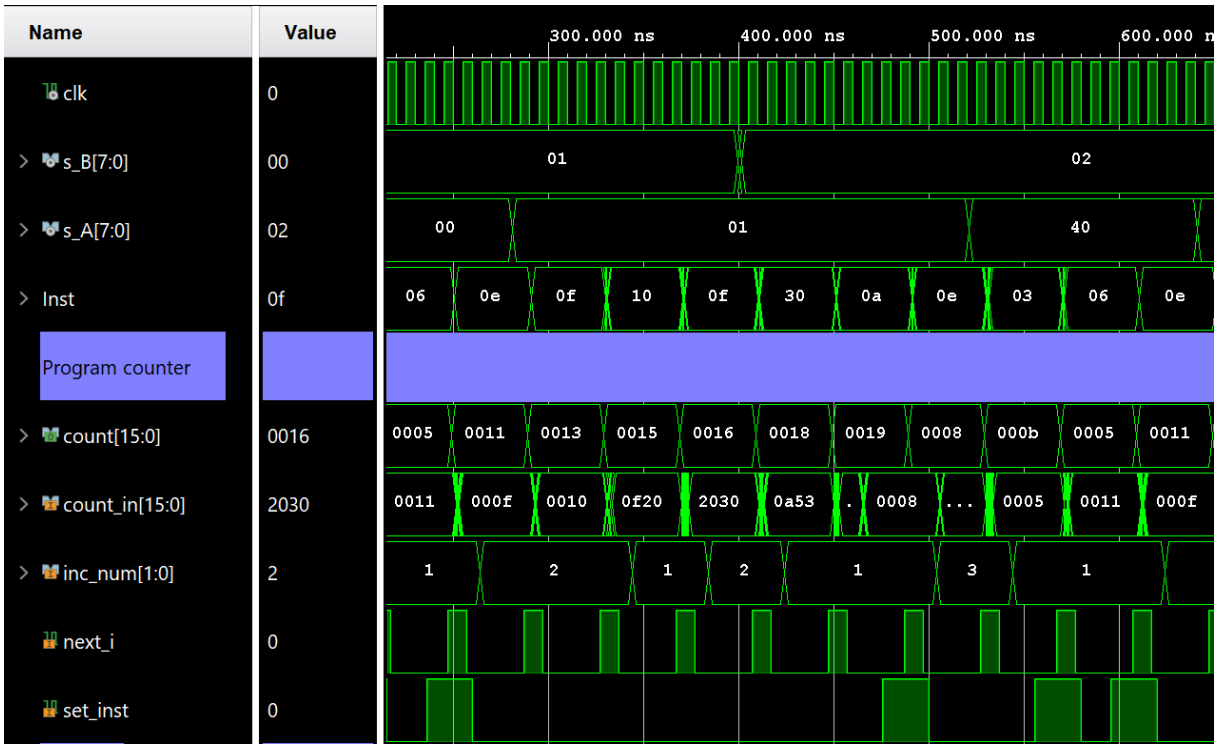
```
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"0F", -- 01 - set B
x"10", -- 02 - from rom
x"01", -- 03 - value
x"30", -- 04 - set reg 0 from B
x"06", -- 05 - jump stuck up
x"00", -- 06 - Add1
x"11", -- 07 - Add2
x"0E", -- 08 - set A
x"10", -- 09 - from rom
x"40", -- 0A - value
x"03", -- 0B - jump if A>B
x"00", -- 0C - Add1
x"05", -- 0D - Add2
x"02", -- 0E - jump
x"00", -- 0F - Add1
x"01", -- 10 - Add2
```

```
INST      ADD  FUNKCE
-- podprogram      ADD -- 11
x"0E", -- 00 - set A
x"00", -- 01 - from reg 0
x"0F", -- 02 - set B
x"00", -- 03 - from reg 0
x"10", -- 04 - ALU +
x"0F", -- 05 - set B
x"20", -- 06 - from ALU
x"30", -- 07 - set reg 0 from B
```



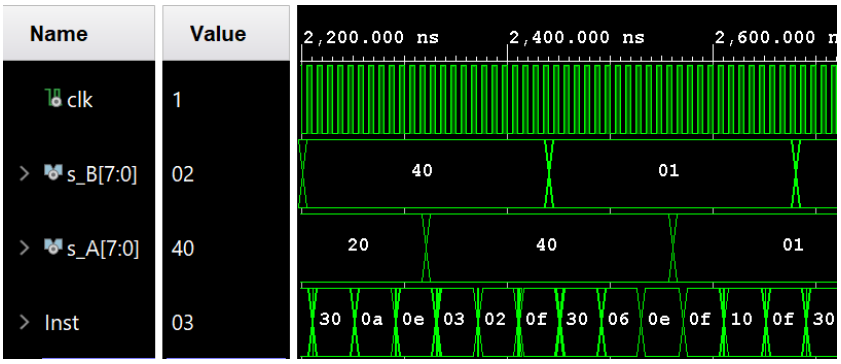
Obrázek 7.5 Program pro výpis geometrické posloupnosti s jeho algoritmem.

Pro jednodušší pochopení je k programu přidán ještě algoritmus, který jednoduše popisuje celé výpočty. Pokud se podíváme na vstupní a výstupní signály programového čítače, tak vidíme, jakým způsobem skoky fungují. Při instrukci, která znamená skok, dostane programový čítač informaci o nastavení instrukce viz obrázek 7.6 set_inst.



Obrázek 7.6 Simulace testování skoku s programovým čítačem

Tento signál je anulován až v následujícím skoku na další instrukci. V tu chvíli se daná hodnota objeví ze sběrnice **count_in** na výstup **count**. Dále je také na této simulaci vidět celá funkce programového čítače při klasickém čítání instrukcí. Vstupní sběrnice do **inc_num** udává délku právě probíhající instrukce, to je pro čítač důležité hlavně z důvodu, aby o tolik následně inkrementoval hodnotu adresy.



Obrázek 7.7 Ukázka reakce na podmínku.

Celý program funguje přesně, jak má. Program vždy skočí do podprogramu, provede výpočet a skočí zpět. Následně ověří, jestli je stále hodnota výsledku menší než hodnota 64, a celý proces opakuje. Ve chvíli, kdy dorazí program k hodnotě 64 reprezentované jako 0x40, přejde zpět na začátek programu a hodnotu vrátí na počáteční hodnotu jedna. Toto dokončení s reakcí na podmínku je vidět na obrázku 7.7.

Celý výpis této posloupnosti od 1 do 64, který odpovídá výpisu a následné kontrole sedmi hodnot, zabere přibližně $2{,}4\ \mu s$. Je jasné, že tento program není optimalizován, jelikož pro provedení takového výpočtu by nebylo potřeba ani registru ani podprogramu. Tím by mohl čas po optimalizaci klesnout na polovinu.

7.4 ČASOVAČ

Pro funkci časovače je nejdůležitější, aby dokázal počítat čas co nejpřesněji. Proto časovač otestuji právě na měření času. V této části vytvořím jednoduchý program, který dokáže počítat čas a vypisovat jeho hodnoty v milisekundách na sběrnici A. Jednoduše řečeno, se bude jednat o proměnnou, která bude inkrementována vždy, když uplyne přesně milisekunda.

Nejprve je potřeba vypočítat nastavení čítače. Budeme uvažovat nad tím, že máme na vstupu 100 MHz oscilátor, což odpovídá 10ns periodě. Abychom docílili požadovaných nanosekund, musíme hodinový signál vydělit 100 000. Jednou z možností je nastavit správce hodin tak, aby na vstup do časovačů pouštěl signál o frekvenci 1 MHz. Tento signál pak ve vstupní děličce stačí vydělit čtyřmi, což následně odpovídá 250 kHz. Následně stačí, aby čítač dopočítal do 250, aby z toho vyšla milisekunda.

Při nastavení již zmíněných hodnot jsem se nejpřesněji dostal na hodnotu 997 µs. K této hodnotě jsem se dostal až ve chvíli, kdy jsem snížil hodnotu, do kolika má čítač napočítat o jedna. Důvodem je, že čítač počítá od nuly. Časový rozdíl oproti požadovanému času dělá rozdíl přibližně 0,3 %. To není až tak velká nepřesnost. Pokud bychom chtěli měřit větší časové úseky, dal by se tento rozdíl vykompenzovat pomocí jednoho čítání, které by bylo o něco naopak delší. Celý program je vidět na obrázku 7.8. V levé části je úvod samotného programu, kde se nastavuje časovač, správce hodin a hodnota na A a B. Následně v pravé části je vidět nekonečný cyklus, který stále dokola přeskakuje na sebe. Pod tímto cyklem je vidět podprogram, který se provede ve chvíli, kdy dojde k dokončení čítání.

```
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"5d", -- 01 - set clk tim
x"30", -- 02 - 200x
x"0F", -- 03 - set B
x"10", -- 04 - from rom
x"FA", -- 05 - value - count in
x"0E", -- 06 - set A
x"10", -- 07 - from rom
x"01", -- 08 - value - div
x"40", -- 09 - set timer 00
x"0F", -- 0a - set B
x"10", -- 0b - from rom
x"1c", -- 0c - value
x"0E", -- 0d - set A
x"10", -- 0e - from rom
x"00", -- 0f - value
x"41", -- 10 - set timer add
x"0F", -- 11 - set B
x"10", -- 12 - from rom
x"01", -- 13 - value
x"0E", -- 14 - set A
x"10", -- 15 - from rom
x"01", -- 16 - value
x"42", -- 17 - timer start
```

```
-- nekonečný cyklus
INST      ADD  FUNKCE
x"53", -- 18 - nic
x"02", -- 19 - jump back
x"00", -- 1a - ADD
x"18", -- 1b - ADD

-- timer
INST      ADD  FUNKCE
x"43", -- 1c - timer rst
x"42", -- 1d - timer start
x"10", -- 1e - ALU +
x"0E", -- 1f - set A
x"20", -- 20 - from ALU
x"02", -- 21 - jump back
x"00", -- 22 - ADD
x"18", -- 23 - ADD
```

Obrázek 7.8 Program nastavování a funkce čítače.

V příloze D naleznete ukázku simulace, na které je ukázána reakce na časovač. Na obrázku je vidět, jak časovač nejprve odešle hodnotu flag a poté následuje vynulování čítače. Pro jistotu je toto vynulování provedeno dvakrát, jednou v rámci flag a podruhé v rámci instrukce reset. Poté je časovač opět spuštěn.

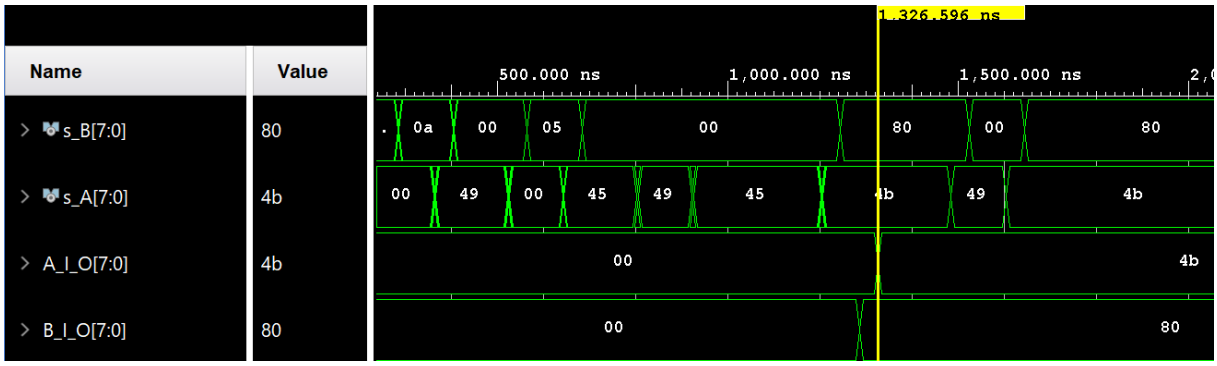
7.5 TESTOVÁNÍ FPU A I/O

Další zkušební program, který jsem si připravil, ukazuje funkci FPU, převodníku mezi formáty unsigned a float a také vstupně výstupní brány. Tento program bude fungovat podobně jako program pro generování geometrické posloupnosti. V tomto případě jsem se ale rozhodl pro jednoduchý program, který bude stále dokola inkrementovat o deset. Pro zajímavější výsledky je počáteční hodnota nastavena na hodnotu 5. Veškeré výsledky budu následně vypisovat na výstupní brány **A** a **B**. Výpis musí být prováděn na dvě brány, jelikož se jedná o 16bitová čísla. Menší problém nastane při reprezentaci čísel na výstupu.

Veškerá čísla, která budou vypsána na brány, budou v zápise float. Z toho vyplývá, že nebudou pro člověka lehce pochopitelná. Pro jednoduší práci s float využiji webovou stránku, která umožňuje jednoduchý překlad hodnoty float tam i zpátky. [17]

Celý program je již celkově dlouhý, jelikož pro ovládání čísel ve formátu float je potřeba velmi mnoho operací navíc. Proto bude celý tento program vložen do přílohy D. Program funguje následovně:

1. Nastavení bran A a B na výstupy.
2. Hodnota 10 se převede do float na hodnotu 0x4900.
3. Uložení této hodnoty do registrů 0000 a 0001.
4. Hodnota 5 se převede do float na hodnotu 0x4500.
5. Uložení této hodnoty do registrů 0010 a 0011.
6. Načtení hodnoty z registrů 0000 a 0001 a uložení do FPU A.
7. Načtení hodnoty z registrů 0010 a 0011 a uložení do FPU B.
8. Sčítání těchto hodnot pomocí FPU.
9. Výsledek uložíme do registrů 0010 a 0011.
10. Výpis výsledku na brány A a B.
11. Skok zpátky na krok 6.

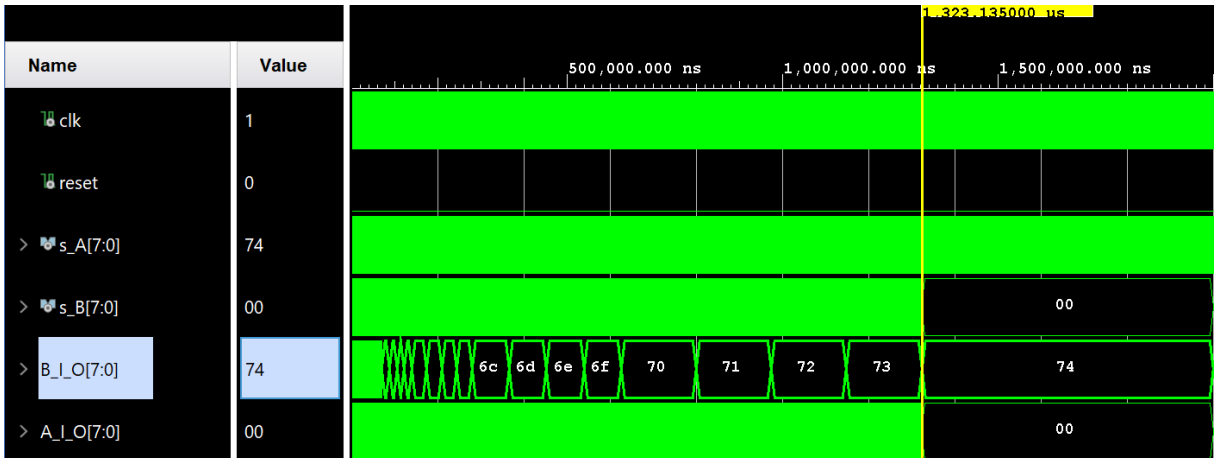


Obrázek 7.9 Simulace výpisu první hodnoty na brány.

Simulace FPU již trvá velmi dlouho, proto na obrázku 7.9 je vidět hlavně simulace výpisu prvních hodnoty na brány. Ze simulace je vidět, že první hodnota, která byla

vypsaná na brány je hodnota 00. Tato hodnota se vždy nastaví jako defaultní hodnota při nastavení brány jako brána výstupní. Následně jsou již vidět první výstupní hodnoty z FPU, dále tedy následuje hodnota 0x4b80. Tato hodnota již reprezentuje číslo 15, což je pravdivý výsledek po sčítání.

Jak již bylo psáno o float, tak se jedná o formát, který daná čísla nezachová v přesné hodnotě, ale snaží se je co nejpřesněji aproximovat. Proto vyvstává otázka, jestli tento program bude fungovat do maximální hodnoty, kterou tento formát dokáže pojmout. Pokud se podíváme na graf, který byl uveden v kapitole 6.6 pro chybu formátu, tak zjistíme, že v jednu chvíli musí nastat situace, kdy chyba formátu převýší přičítanou hodnotu. V takovou chvíli se již nedokáže hodnota 10 přičíst a zůstane stále stejná. Pro test, kdy k takové situaci dojde, použiji behaviorální simulaci, která funguje velmi dobře i pro delší intervaly simulací.



Obrázek 7.10 Ukázka behaviorální simulaci při překročení chyby float.

Maximální hodnota, do které se dá dopočítat inkrementací hodnoty 10, je výsledná hodnota 0x7400, tato hodnota odpovídá číslu 16384. Důvodem, proč již nelze přičíst více, je ten, že následující hodnota 0x7401 již odpovídá hodnotě 16400, což je o 16 větší hodnota. Další zajímavostí, která je na tomto čísle vidět, je, že již neodpovídá žádné hodnotě v této posloupnosti, kdy přičítáme číslo 10. Důvodem je opět nepřesnost tohoto formátu, kdy již nelze přičíst číslo 10, ale jen číslo, které je mocninou čísla 2.

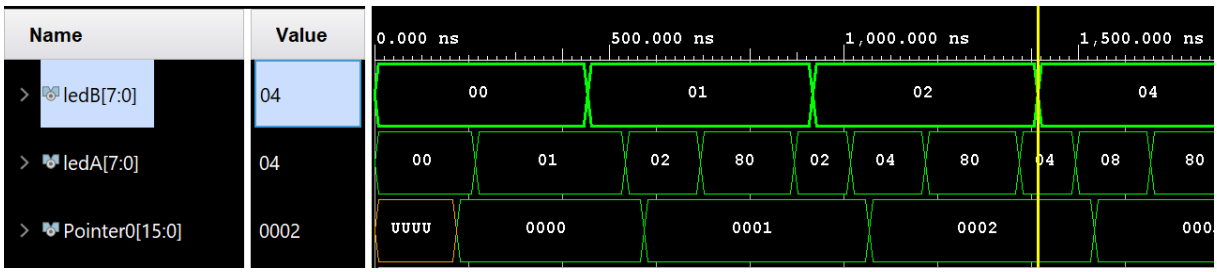
Otázka tedy zní, do jaké hodnoty lze ještě výsledku věřit. Poslední přesná hodnota se nachází mnohem dříve. Jedná se o hodnotu 0x67fd, která odpovídá hodnotě 2045. Důvodem je, že se jedná o poslední hodnotu, která podporuje přičítání hodnoty jedna, a proto lze ještě vypsat hodnotu 5 na konci. Následující hodnoty budou mít již vždy na konci hodnotu 4 místo hodnoty 5

7.6 PROGRAM K TESTOVÁNÍ RAM

Jak již bylo napsáno v kapitole o pamětech, které se nachází v tomto MCU, tak jsem pro ukládání dat vytvořil dvojici pamětí. V kapitole 7.2 jsme testovali funkčnost registrů, které slouží k velmi rychlému přístupu k datům. V tomto programu otestujeme funkčnost bloku RAM a přiřazeného bloku, který spravuje pointery k paměti.

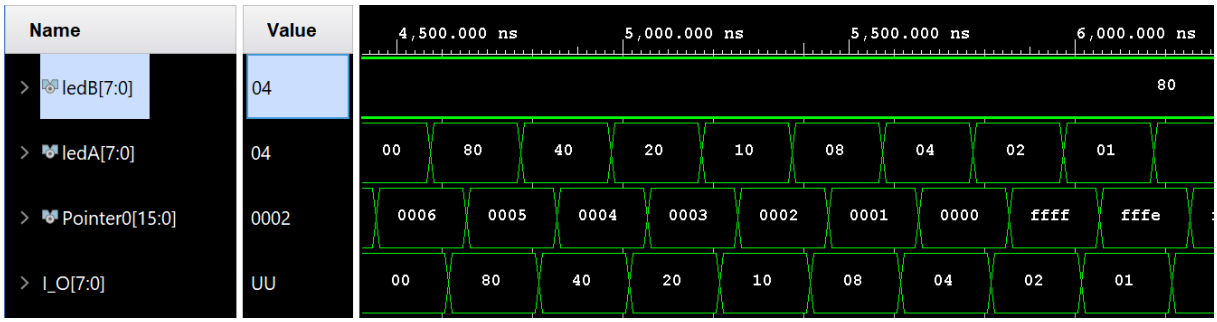
Veškeré adresy do paměti RAM jsou řízeny pomocí pointerů, což umožňuje jednoduchou inkrementaci a dekrementaci adres. V tomto programu bude právě ukázáno, jak tento systém funguje a k čemu se dá využít. Pro ukázku jsem zvolil program, který je velmi podobný programu z kapitoly 7.3. Tentokrát ale půjde o geometrickou posloupnost, která se bude vypisovat v obráceném pořadí. To znamená, že na výstupu se budou zobrazovat hodnoty od nejvyšší po nejmenší.

Tento program bude fungovat ve dvou krocích. V prvním kroku bude geometrická posloupnost generována stejně jako v již zmíněné kapitole. To znamená, že hodnoty budou opět tvořeny pomocí sčítání. Veškeré výsledky posloupnosti budou následně ukládány do pole dat, které bude vytvářeno právě pomocí inkrementace pointeru. V druhém kroku budou potřeba data vypsat na výstupní bránu A. K tomu opět využijeme pointer, který se bude naopak každý krok dekrementovat. Pro jistotu, aby nedošlo k přetečení, budou hodnoty generovány jen do hodnoty 0x80. Důvodem je, že následující hodnota by dosáhla opět hodnoty 0x00 kvůli již zmíněnému přetečení.



Obrázek 7.11 Část generování dat a ukládání do RAM.

Celý program je ukázán v příloze D. Tento program je rozdělen do tří částí. V první části jde především o nastavení výchozích hodnot, nastavení brány A na výstup a připravení bloku s pointery. Dále program pokračuje nekonečnou smyčkou, která zajistí kontrolu, zda je hodnota stále pod požadovanou hodnotou. Pokud ano, tak program přeskočí do druhé části. Druhou částí je podprogram, který se stará o načtení hodnot z RAM a jejich sečtení. Dále program inkrementuje hodnotu pointeru a tuto adresu uloží výsledek po sčítání. Takto program pokračuje, dokud se nedostane do hodnoty 0x80. Celý jako tento proces je vidět na obrázku 7.11 , kde je na simulaci vidět i hodnota pointeru. Po této hodnotě program přejde do třetí části, kterou je výpis.



Obrázek 7.12 Výpis hodnot na bránu A.

Výpis funguje podobně jako při ukládání hodnot. Rozdílem je, že se pointer v RAM zmenšuje a tím se postupně vrací na předchozí hodnoty. Po každé dekrementaci jsou data uložena na bránu A. Tím je také dosaženo toho, že jsou hodnoty vypisovány mnohem rychleji, než kdybychom chtěli hodnoty přímo počítat a vypisovat je po každém výpočtu. Celý výpis je opět vidět na obrázku 8.12, kde je vidět simulace výpisu hodnot. Dále je také na obrázku vidět, že na konci simulace dojde k přetečení hodnot pointeru. Tím je způsobeno, že program na konci nemá zadanou žádnou konečnou hodnotu. Kvůli tomu tedy program pokračuje i po dokončení výpisu všech hodnot.

8 ZÁVĚR

V rámci tohoto projektu jsem si v mnohém osvojil fungování mikrokontroléru a jeho funkčních bloků. První část práce se zaměřuje na historii mikrokontrolérů a dále vysvětlila, jak samotný mikrokontrolér funguje a jaké funkční bloky obsahuje. Poté jsem se zaměřil na FPGA a jeho funkční bloky, s důrazem na architekturu Artix7, na které byl následně vystavěn mikrokontrolér. Jelikož veškeré výsledky byly získány převážně pomocí simulací, věnoval jsem část práce prostředí Vivado, ve kterém byl mikrokontrolér stavěn a simulován.

Praktická část byla zaměřena na samotnou stavbu mikrokontroléru. Každá kapitola se věnovala realizaci různých bloků, včetně ALU, ROM, programového čítače, zásobníku, registrů, vstupně-výstupních bran, časovačů, FPU, převodníku mezi Float a unsigned, RAM, řídící jednotky a hodinového managementu. Tyto bloky jsou podrobně popsány pomocí blokových schémat. Dále je v této části vysvětlena celková funkčnost a programová sada. Výsledkem této části je funkční mikrokontrolér, který dokáže běžet až na 100 MHz. MCU je dostatečně popsán pro jeho případné rozšíření o další bloky, a může tak sloužit jako zdroj informací pro studenty, kteří se chtějí dozvědět více o fungování mikrokontroléru.

Třetí část se věnovala programům, které ukazují funkčnost všech bloků. Tyto programy byly ověřeny pomocí post-implementační simulace, která dobře ukazuje, jak by se toto zapojení chovalo na FPGA. Programy jsou dostatečně popsány, aby s nimi mohlo být snadné pracovat. Vznikla tak šestice programů, které postupně ovládají veškeré komponenty mikrokontroléru.

Tento mikrokontrolér má dostatek volné programové paměti pro vytvoření dalších užitečných bloků, například pro spravování priority přerušení nebo pro komunikaci pomocí I2C. Lze také vytvořit bloky pro řízení PWM.

Srovnáním zadání mé diplomové práce s dosaženými výsledky a výsledky prezentovanými v tomto závěrečném souhrnu lze konstatovat, že jsem úspěšně zvládl a naplnil veškeré požadavky. Navržený mikrokontrolér (MCU) je plně funkční a může být efektivně aplikován v mnoha praktických situacích. Kompletní dokumentace týkající se navrženého MCU, včetně vlastních VHDL kódů a zdrojových kódů mikrokontroléru, je součástí elektronické přílohy této diplomové práce.

BIBLIOGRAFIE

[1] Mahajan, Mukesh & Aswale, Pramod & Ugale, Vivek. (2014). FPGA Implementation of ARM Processor. IJAREEIE. 3297. 2320-3765. 10.15662/ijareeie.2014.0311026.

[2] Betker, M.R., Fernando, J.S. and Whalen, S.P. (1997), The history of the microprocessor. Bell Labs Tech. J., 2: 29-56. https://doi.org/10.1002/bltj.2082

[3] Intel-4004: single chip 4-bit [online]. 23 [cit. 2023-01-08]. Dostupné z: http://datasheets.chipdb.org/Intel/MCS-4/datashts/intel-4004.pdf

[4] YADAV, D.S. a Arun Kumar SINGH. Microcontrollers: features and applications. New Delhi: New Age International, 2004, 267 s. ISBN 9798122415697.

[5] STEINER, Craig. *The 8051/8052 microcontroller: architecture, assembly language, and hardware interfacing*. Universal-Publishers, 2005.

[6] JIANG, Jean. Digital Signal Processing: Fundamentals and Applications. 2nd. New Mexico: Academic Press, 2013. ISBN 978-0124158931.

[7] S. D. Trong, M. Schmookler, E. M. Schwarz and M. Kroener, "P6 Binary Floating-Point Unit," 18th IEEE Symposium on Computer Arithmetic (ARITH '07), 2007, pp. 77-86, doi: 10.1109/ARITH.2007.26.

[8] WITTENBRINK, Craig M.; KILGARIFF, Emmett; PRABHU, Arjun. Fermi GF100 GPU architecture. *IEEE Micro*, 2011, 31.2: 50-59.

[9] DAVIES, John H. *MSP430 microcontroller basics*. Elsevier, 2008.

[10] WANG, Haibo. CPLD and FPGA Architectures [online]. In: . Southern Illinois: University Carbondale [cit. 2023-03-01]. Dostupné z: https://www.engr.siu.edu/haibo/ece428/notes/ece428_fpgaarch.pdf

[11] KAITLYN Franz History of the FPGA [online] [cit. 2023-03-01]. Dostupné z: https://blog.digilentinc.com/history-of-the-fpga/

[12] *7 Series FPGAs Overview: Advance Product Specification* [online]. November 30, 2012, 16 [cit. 2023-02-06]. Dostupné z: https://www.xilinx-adm.com/files/ed/XC7K355T-2FFG901C.pdf

[13] *7 Series FPGAs Configurable Logic Block: User Guide* [online]. September 27, 2016, 74 [cit. 2023-02-06]. Dostupné z: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB

[14] *7 Series DSP48E1 Slice: User Guide* [online]. March 27, 2018, 58 [cit. 2023-02-06]. Dostupné z: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[15] *7 Series DSP48E1 Slice: User Guide* [online]. July 30, 2018, 114 [cit. 2023-02-07]. Dostupné z: https://docs.xilinx.com/v/u/en-US/ug472_7Series_Clocking

[16] *7 Series FPGAs Memory Resources: User Guide* [online]. July 3, 2019, 88 [cit. 2023-02-07]. Dostupné z: https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources

[17] Float Toy. *Float Toy* [online]. [cit. 2023-03-12]. Dostupné z: https://evanw.github.io/float-toy/

PŘÍLOHA A

SEZNAM ZKRATEK

- ALU - Aritmetic Logic Unit
- ASIC - Application Specific Integrated Circuit
- BLE - Basic Logic Element
- CLBs - Configurable Logic Blocks
- CMTs - Clock Managment Tiles
- CISC - Complex Instruction Set Computer
- CPLD - Complex Programmable Logic Device
- CPU - Central Processing Unit
- CU - Controll unit
- DSP - Digital Signal Processing
- DTFT - Discrete-time Fourier transform
- FFT - Fast Fourier Transform
- FIFO - First In First Out
- FILO - First In Last Out
- FPGA - Field Programmable Gate Array
- FPU - Floating point unit
- HDR - Hardware Description Language
- I/O - Input/Output
- MCU - Microcontroller
- MMCM - Mixed-mode Clock Manager
- MUX - Multiplexor
- PAL - Programmable Array Logic
- PLD - Programmable Logic Devices
- PLL - Phase-locked Loop
- PROM - Programmable Read Only Memory
- PWM - Pulse-width modulation
- RAM - Random Access Memory
- ROM - Read Only Memory
- RISC - Reduced Instruction Set Computer
- VHDL - VHSIC Hardware Description Language
- VHSIC - Very High Speed Integrated Circuit
- µP - Microprocessor

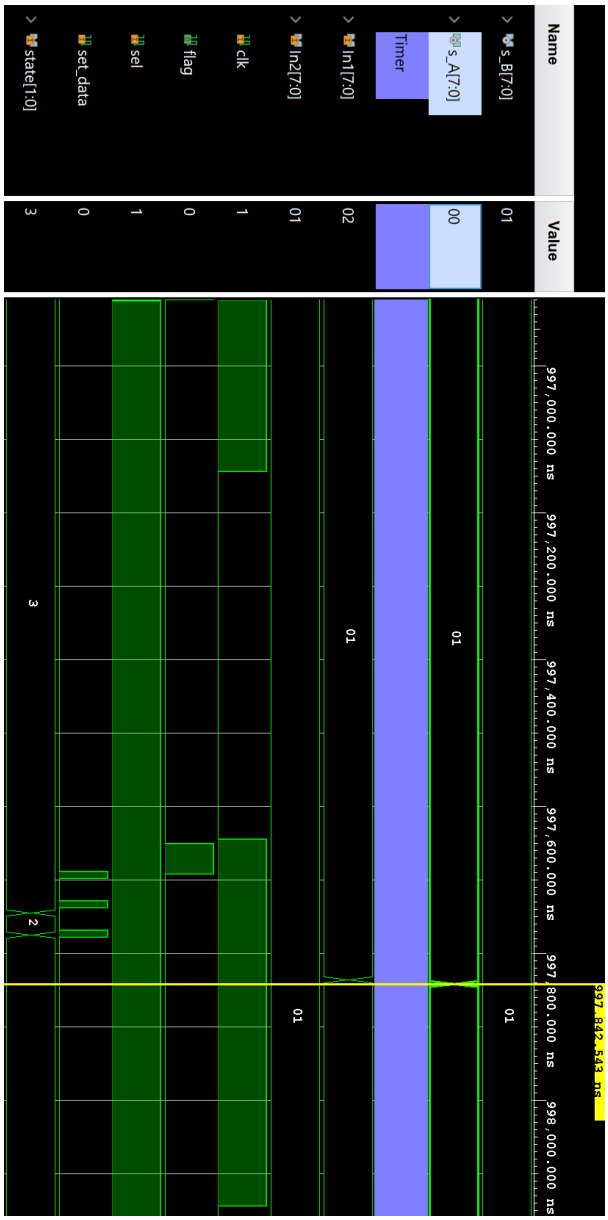
PŘÍLOHA C

INSTRUKČNÍ SADA

1 B	2 B	Funkce
00000000		Restart celého MCU
00000001	SSMMI000	Nastavení I/O, kde SS – adresa Brány, MM – mód funkce, I – Vstupní sběrnice.
00000010	AAAAAAAA	Skok na adresu z následujících dvou bajtů.
00000011	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud A>B.
00000100	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud A=B.
00000101	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud Carryout.
00000110	AAAAAAAA	Skok na adresu z následujících dvou bajtů. Uložení aktuální adresy do zásobníku
00000111	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud A>B. uložení aktuální adresy do zásobníku
00001000	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud A=B. uložení aktuální adresy do zásobníku
00001001	AAAAAAAA	Skok na adresu z následujících dvou bajtů, pokud Carryout. Uložení aktuální adresy do zásobníku
00001010		Skok na adresu zásobníku.
00001011		Skok na adresu zásobníku, pokud A>B.
00001100		Skok na adresu zásobníku, pokud A=B.
00001101		Skok na adresu zásobníku, pokud Carryout.
00001110	0SSSAAAA	Nastavení sběrnice A. SSS – odkud (000 = Reg, 001=ROM, 010=ALU, 011=I/O, 100=I_to_F, 101=FPU, 110=RAM). AAAA-adresa registru
00001111	0SSSAAAA	Nastavení sběrnice B. SSS – odkud (000 = Reg, 001=ROM, 010=ALU, 011=I/O, 100=I_to_F, 101=FPU, 110=RAM). AAAA-adresa registru
0001SSSS		Ovládání ALU, kde SSSS jsou jednotlivé instrukce.
0010AAAA		Uložení do registru AAAA ze sběrnice A
0011AAAA		Uložení do registru AAAA ze sběrnice B
0100SSAA		Nastavení časovače AA do módu SS
01010000		Převod unsigned na Float
01010001		Převod Float na Unsigned
01010010		Nastavení vstupních data do FPU
01010011		Nulová instrukce
010101SS		Výpočet pomocí FPU. SS – operace
01011000		Zápis do RAM na pointer ze sběrnice A
01011001		Zápis do RAM na pointer ze sběrnice B
01011010		Načtení hodnoty z RAM
01011011	SFAVAAAA	Nastavení pointeru. S-pouze selekce, F – ALU funkce, A – mux in, V- nastavení hodnoty Value, AAAA-adresa pointeru.
01011100	DDDDDDDD	Nastavení hodin pro řídící jednotku na hodnotu DDDDDDDD
01011101	DDDDDDDD	Nastavení hodin pro časoveče na hodnotu DDDDDDDD
01011110	DDDDDDDD	Nastavení hodin pro FPU na hodnotu DDDDDDDD

PŘÍLOHA D

SIMULACE ČASOVAČE



TEST ALU

```
INST      ADD  FUNKCE   INST      ADD  FUNKCE
x"00", -- 01 - reset    x"10", -- 0C - from rom
x"0E", -- 02 - set A    x"05", -- 0D - value
x"10", -- 03 - from rom x"11", -- 0E - ALU -
x"0a", -- 04 - value    x"0E", -- 0F - set A
x"0F", -- 05 - set B    x"20", -- 10 - from ALU
x"10", -- 06 - from rom x"0F", -- 11 - set B
x"02", -- 07 - value    x"10", -- 12 - from rom
x"12", -- 08 - ALU *    x"03", -- 13 - value
x"0E", -- 09 - set A    x"13", -- 14 - ALU /
x"20", -- 0A - from ALU x"0E", -- 15 - set A
x"0F", -- 0B - set B    x"20", -- 16 - from ALU
```

TEST REGISTRŮ

```
INST      ADD  FUNKCE
x"00", -- 01 - 0
x"0E", -- 02 - set A
x"10", -- 03 - from rom
x"03", -- 04 - value
x"0F", -- 05 - set B
x"10", -- 06 - from rom
x"05", -- 07 - value
x"12", -- 08 - ALU *
x"0E", -- 09 - set A
x"20", -- 0A - from ALU
x"20", -- 0B - save A to reg 0000
x"0E", -- 0C - set A
```

```
INST      ADD  FUNKCE
x"10", -- 0D - from rom
x"04", -- 0E - value
x"0F", -- 0F - set B
x"10", -- 10 - from rom
x"02", -- 11 - value
x"12", -- 12 - ALU *
x"0E", -- 13 - set A
x"20", -- 14 - from ALU
x"0F", -- 15 - set B
x"00", -- 16 - from reg 0000
x"10", -- 17 - ALU +
x"0E", -- 18 - set A
x"20", -- 19 - from ALU
```

TEST PRO PODMÍNĚNÉ SKOKY

```
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"0F", -- 01 - set B
x"10", -- 02 - from rom
x"01", -- 03 - value
x"30", -- 04 - set reg 0 from B
x"06", -- 05 - jump stuck up
x"00", -- 06 - Add1
x"11", -- 07 - Add2
x"0E", -- 08 - set A
x"10", -- 09 - from rom
x"40", -- 0A - value
x"03", -- 0B - jump if A>B
x"00", -- 0C - Add1
x"05", -- 0D - Add2
x"02", -- 0E - jump
x"00", -- 0F - Add1
x"01", -- 10 - Add2
```

```
INST      ADD  FUNKCE
-- podprogram      ADD -- 11
x"0E", -- 00 - set A
x"00", -- 01 - from reg 0
x"0F", -- 02 - set B
x"00", -- 03 - from reg 0
x"10", -- 04 - ALU +
x"0F", -- 05 - set B
x"20", -- 06 - from ALU
x"30", -- 07 - set reg 0 from B
```

TEST ČASOVAČE

```
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"5d", -- 01 - set clk tim
x"30", -- 02 - 200x
x"0F", -- 03 - set B
x"10", -- 04 - from rom
x"FA", -- 05 - value - count in
x"0E", -- 06 - set A
x"10", -- 07 - from rom
x"01", -- 08 - value - div
x"40", -- 09 - set timer 00
x"0F", -- 0a - set B
x"10", -- 0b - from rom
x"1c", -- 0c - value
x"0E", -- 0d - set A
x"10", -- 0e - from rom
x"00", -- 0f - value
x"41", -- 10 - set timer add
x"0F", -- 11 - set B
x"10", -- 12 - from rom
x"01", -- 13 - value
x"0E", -- 14 - set A
x"10", -- 15 - from rom
x"01", -- 16 - value
x"42", -- 17 - timer start
```

```
-- nekonečný cyklus
INST      ADD  FUNKCE
x"53", -- 18 - nic
x"02", -- 19 - jump back
x"00", -- 1a - ADD
x"18", -- 1b - ADD

-- timer
INST      ADD  FUNKCE
x"43", -- 1c - timer rst
x"42", -- 1d - timer start
x"10", -- 1e - ALU +
x"0E", -- 1f - set A
x"20", -- 20 - from ALU
x"02", -- 21 - jump back
x"00", -- 22 - ADD
x"18", -- 23 - ADD
```

PROGRAM PRO FPU

```
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"01", -- 01 - set I/O
x"10", -- 02 - mode out A
x"01", -- 03 - set I/O
x"50", -- 04 - mode out B
x"0E", -- 05 - set A
x"10", -- 06 - from rom
x"00", -- 07 - value
x"0F", -- 08 - set B
x"10", -- 09 - from rom
x"0a", -- 0a - value
x"50", -- 0b - I to F
x"0E", -- 0c - set A
x"40", -- 0d - from I to F
x"0F", -- 0e - set B
x"40", -- 0f - from I to F
x"20", -- 10 - set reg 0 from A
x"31", -- 11 - set reg 1 from B
x"0E", -- 12 - set A
x"10", -- 13 - from rom
x"00", -- 14 - value
x"0F", -- 15 - set B
x"10", -- 16 - from rom
x"05", -- 17 - value
x"50", -- 18 - I to F
x"0E", -- 19 - set A
x"40", -- 1a - from I to F
x"0F", -- 1b - set B
x"40", -- 1c - from I to F
x"22", -- 1d - set reg 2 from A
x"33", -- 1e - set reg 3 from B
```

```
-- opakování
INST      ADD  FUNKCE
x"0E", -- 1f - set A
x"00", -- 20 - from reg 0
x"0F", -- 21 - set B
x"01", -- 22 - from reg 1
x"52", -- 23 - FPU set data
x"0E", -- 24 - set A
x"02", -- 25 - from reg 2
x"0F", -- 26 - set B
x"03", -- 27 - from reg 3
x"52", -- 28 - FPU set data
x"54", -- 29 - FPU sčítání
x"53", -- 2a - nic
x"53", -- 2b - nic
x"53", -- 2c - nic
x"0E", -- 2d - set A
x"50", -- 2e - from FPU
x"0F", -- 2f - set B
x"50", -- 30 - from FPU
x"01", -- 31 - set I/O
x"30", -- 32 - mode out A form A
x"01", -- 33 - set I/O
x"78", -- 34 - mode out B form B
x"22", -- 35 - set reg 2 from A
x"33", -- 36 - set reg 3 from B
x"02", -- 37 - Jump to
x"00", -- 38 - add1
x"1f", -- 39 - add2
```

PROGRAM PRO RAM

```
-- vypis na konci
INST      ADD  FUNKCE
x"5a", -- 18 - read form RAM
x"0E", -- 19 - set A
x"60", -- 1a - from ram
x"01", -- 1b - set I/O
x"38", -- 1c - mode out
x"5b", -- 1d - set pointer
x"60", -- 1e - dekrementace
x"02", -- 1f - jump add if A > B
x"00", -- 20 - add
x"18", -- 21 - add

--        22 - podprogram
x"5b", -- 00 - set pointer
x"80", -- 01 - only set
x"5a", -- 02 - read form RAM
x"0E", -- 03 - set A
x"60", -- 04 - from ram
x"0F", -- 05 - set B
x"60", -- 06 - from ram
x"10", -- 07 - ALU +
x"0E", -- 08 - set A
x"20", -- 09 - from ALU
x"5b", -- 0a - set pointer
x"20", -- 0b - ++
x"58", -- 0c - wrte from A
x"0a", -- 07 - jump back
```

```
-- RAM
INST      ADD  FUNKCE
x"00", -- 00 - 0
x"01", -- 01 - set I/O
x"10", -- 02 - mode out
x"0E", -- 03 - set A
x"10", -- 04 - from rom
x"00", -- 05 - value
x"0F", -- 06 - set B
x"10", -- 07 - from rom
x"00", -- 08 - value
x"5b", -- 09 - set pointer
x"00", -- 0a - form AB
x"0E", -- 0b - set A
x"10", -- 0c - from rom
x"01", -- 0d - value
x"58", -- 0e - RAM from A
x"06", -- 0f - jump add stack up
x"00", -- 10 - add
x"22", -- 11 - add
x"0E", -- 12 - set A
x"10", -- 13 - from rom
x"80", -- 14 - value
x"03", -- 15 - jump add if A > B
x"00", -- 16 - add
x"0f", -- 17 - add
```

PŘÍLOHA E

PŘÍLOHY

- Schematic
  - ALU.pdf
  - Control_unit.pdf
  - FPU.pdf
  - Int_to_F.pdf
  - IO.pdf
  - MUC.pdf
  - Program_counter.pdf
  - RAM.pdf
  - Stack.pdf
  - Time_Manegment.pdf
  - Timer.pdf
- 8bit_MCU
  - 8bit_MCU.gen
  - 8bit_MCU.hw
  - 8bit_MCU.srcs
  - 8bit_MCU.xpr
  - Nexys-4-Master.xdc