ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA ELEKTROTECHNICKÁ

BAKALÁŘSKÁ PRÁCE

FPGA Demonstrátor soft mikroprocesoru

Praha, 2015

Autor práce: **Adam Patera**
Vedoucí práce: **Ing. Stanislav Vítek, Ph.D.**

Čestné prohlášení

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

V Praze dne ....

Podpis

Poděkování

Tímto bych rád poděkoval Ing. Stanislavovi Vítkovi, Ph.D. za odborné vedení a cenné rady,které mi ochotně poskytoval při vypracovávání mé bakalářské práce.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra mikroelektroniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **P A T E R A Adam**

Studijní program: Komunikace, multimédia a elektronika
Obor: Aplikovaná elektronika

Název tématu: **FPGA demonstrátor soft mikroprocesoru**

Pokyny pro vypracování:

1) Navrhněte koncepci syntetizovatelného soft-mikroprocesoru.
2) Součástí práce bude návrh redukované instrukční sady.
3) Navržený soft-procesor implementujte na FPGA.
4) Možnosti procesoru demonstrujte na příkladech.

Seznam odborné literatury:

[1] Pedroni, V. A., Circuit Design and Simulation with VHDL, MIT Press, 2010, ISBN 978 -0262014335
[2] Pedroni, V. A., Finite State Machines in Hardware, MIT Press, 2013, ISBN 978 -0262019668
[3] Šťastný J., Bílý P., Návrh mikrořadiče na FPGA, Elektrorevue, No. 30, 2006 [dostupné online: http://www.elektrorevue.cz/clanky/06030/index.html]

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

Platnost zadání: 31. 8. 2016

L.S.

prof. Ing. Miroslav Husák, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 2. 2. 2015

Abstrakt

Bakalářská práce rozebírá návrh koncepce syntetizovatelného soft-mikroprocesoru na architektuře FPGA. Programovatelná hradlová pole FPGA jsou v mnoha ohledech přelomovou technologií a na trhu mají své nezastupitelné místo. Převážná většina textů se přirozeně soustředí na vývoj specializovaných (dedikovaných) obvodů a aplikací, které mohou těžit z masivního paralelismu programovatelných hradel. Následující stránky jsou v opačném duchu věnovány funkčnímu popisu tradičních sekvenčních zařízení známého z mikroprocesorové techniky. Toto přiměřeně rozsáhlé a zároveň klasické téma umožňuje obecné seznámení s principy digitální logiky a jazyky hardwarového popisu (HDL). Nabyté vědomosti jsou pak přenosné i do podstatně složitějších aplikací.
Celková funkčnost soft-mikroprocesoru byla v závěru důkladně ověřena prostřednictvím ukázkových programů a vybraných simulací. Součástí práce jsou rovněž kompletní komentované HDL zdrojové kódy implementovaného zařízení v příloze.

Klíčová slova: Soft-mikroprocesor, FPGA, VHDL, RISC

Abstract

This Bachelor thesis discusses the conceptual design of synthetizable soft-microprocessor implemented using Field Programmable Gate Array (FPGA) architecture. FPGAs are in many aspects breakthrough technology and as such have irreplacable position on the market. Most available articles naturally focuses on developing real-life dedicated circuits and applications, which are able to benefit from massive parallelism offered by the FPGAs. Following pages, on the other hand, are devoted to functional description of traditional sequential devices already known from the field of microprocessor technology. This adequately large but classic topic allows for general understanding of digital logic and Languages of Hardware Description (HDL) to be made. Acquired knowledge are then transferable into substantially advanced applications.
In conclusion is the overall functionality of the design thoroughly verified and tested by means of sample programs and simulations. Complete HDL description, along with proper comments, of the device can be found in the attachment of the thesis.

Keywords: Soft-microprocessor, FPGA, VHDL, RISC

Obsah

Kapitola 1

Úvod

1.1 Předmluva

Programovatelná logická zařízení, označovaná plným názvem jako *Programmable Logic Devices* (*PLD*), umožňují pohodlnou tvorbu konfigurovatelných digitálních obvodů. Prostřednictvím moderních PLD je možné realizovat libovolnou logickou funkci či sekvenční obvod.
První programovatelná zařízení byla na trh uvedena již v 70. letech 20. století. Typickými zástupci PLD technologie se tak staly obvody PLA (*Programmable Logic Array*) a PAL (*Programmable Array Logic*). PLA i PAL obsahovala rozsáhlou matici tradičních logických hradel logického součinu AND a logického součtu OR. Vzájemným propojením velkého počtu hradel bylo možno zapsat jakoukoliv logickou funkci v disjunktní normálové formě, tedy součtů součinů jednotlivých midtermů. Tyto ranné PLD obvody byly však vhodné pouze ke konstrukci kombinační logiky a to z důvodu totální absence paměťových prvků.
Toto omezení bylo odstraněno s nástupem obvodů GAL (*Generic Array Logic*), které nově obsahovaly vedle existující matice hradel i množství paměťových buňek (*flip flopů*) a multiplexorů. Kombinační a sekvenční logika byla sloučena do tzv. makrobuněk (*macrocell*). Výstupy mohly býti pomocí multiplexorů směřovány do sousedních či vlastních makrobuněk, což otevřelo možnosti dosud nevídané. Kombinace velkého počtu PLA, PAL a GAL pak vešla do paměti jako kategorie zařízení CPLD (*Complex Programmable Logic Devices*). Paralelně s CPLD také došlo k představení FPGA (*Field Programmable Gate Arrays*).

Základní stavební jednotkou jednotkou FPGA je CLB (*Configurable Logic Block*). Zde je namístě podotknout, že značení a do jisté míry i provedení komponent se může lišit v závislosti na výrobci. CLB dále obsahuje speciální paměťové buňky LUT (*Look-up Table*) a množství flip flopů a multiplexorů. LUT je principiálně paměťová SRAM buňka, která může sestávat například z dvojice n-mos invertorů a

přístupových tranzistorů. Otvírání a zavírání přístupových tranzistorů kontroluje zápis do buňky. SRAM jsou slučovány tradičně do bloku po 16-ti, pro adresování dané buňky je tak ještě zapotřebí 4:16 dekodéru. Takto je zvolena vždy jedna buňka, ostatní buňky jsou ve stavu vysoké impedance. Programování FPGA tak znamená programování velkého množství LUT komponent. Po naprogramování má již LUT čistě charakter kombinační logiky, neboť do buňky není dále zapisováno – zápis je jednorázový. Můžeme říci, že LUT obsahuje pravdivostní tabulku – logickou funkci. Aby bylo možno na FPGA realizovat i sekvenční logiku, musí mít zařízení prostředky pro uchování současného stavu (výstupu). Výstup LUT je tak veden do flip flopu a navíc ještě do multiplexoru, jehož druhým vstupem je právě výstup z tohoto flip flopu. Typický LUT pak disponuje čtyřmi či šesti vstupy a právě jedním výstupem.

K popisu programovatelných digitálních obvodů se používají jazyky hardwarového popisu HDL (*Hardware Description Language*), kterých existuje celá řada. Jmenujme alespoň dva dominantní zástupce a to VHDL (*VHSIC Hardware Description Language – Very High Speed Integrated Circuit Hardware Description Language*) a Verilog. Všechny dodané zdrojové kódy, včetně výňatků zde v textu uvedené, jsou psány v jazyce VHDL.

1.2 Účel práce

Předmětem práce byl návrh syntetizovatelného soft-mikroprocesoru včetně redukované instrukční sady. V textu jsou postupně rozebrány veškeré stavební bloky, logika a další podrobnosti implementovaného mikroprocesoru. Cílem bylo realizovat přiměřeně složité a účelové zařízení, které by přispělo k rozvoji porozumnění základního funkčního principu mikroprocesorové a digitální techniky obecně. Práce tak rovněž nabízí hezký úvod do jazyka VHDL prostřednictvím přiložených komentovaných zdrojových kódů mikroprocesoru.

1.3 Poznámky k notaci a členění práce

Text je intuitivně členěn do kapitol a oddílů, které popisují jednotlivé komponenty mikropočítače v pořadí daném smyslem Obr.1 níže. V oddílech druhé kapitoly jsou tak postupně rozebrány význam a prostředky datové cesty následované oddílem o řídících obvodech. Předmětem třetí kapitoly textu jsou pak základní integrované periférie. Ve čtvrté kapitole můžeme najít vyčerpávající přehled a definice všech dostupných instrukcí. Instrukční sadě je tedy z důvodu patřičné názornosti věnována samostatná kapitola, ačkoliv svým charakterem ji zcela jistě můžeme zařadit do kapitoly druhé.
Konečně je v páté kapitole mikropočítač podroben zkoušce, na které je prokázána funkčnost celého návrhu. V závěrečné kapitole pak mimojiné nalezneme krátké zamyšlení nad možnými úpravami a dalšími možnostmi mikropočítače.

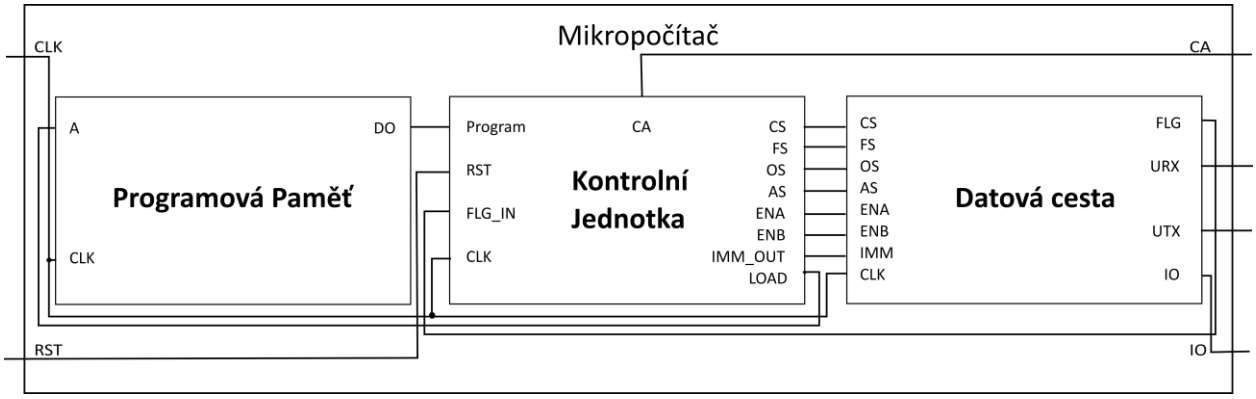
V krátkosti ještě zmíníme několik postřehů týkajícího se použitého značení v textu:

- Všechna bitová pole instrukcí, registrů, paměťových slov apod. jsou číslována od LSB (*Least Significant Bit*) do MSB (*Most Significant Bit*), kde LSB se vyskytuje vždy na pravé straně bitové posloupnosti a MSB na straně levé. Mikropočítač tedy respektuje přirozený *Little Endian* standard.
- V textu jsou výhradně používána čísla o základu 16 (hexadecimální) a 2 (binární). Hexadecimální čísla zde značíme předponou „0x“, příkladem budiž „0x123“. Binární čísla odlišujeme příponou „b“, tedy například „111100001111b“. V instrukčních definicích tuto notaci z estetických příčin vynecháváme, posloupnost instrukčních polí je vždy zapsána binárně. Adresa „AAAAAAAAAAAA“ tudíž reprezentuje plné 12-ti bitové číslo a „BB“ 2-bitové číslo.
- Každá existující instrukce má svůj mnemonický přepis a lze ji zapsat pomocí jazyka symbolických adres ve formátu *mnemonika* $operand_1$, $operand_2$. *Mnemonika* je symbolické označení operačního kódu (opkódu) instrukce. Počet operandů je volitelný a závisí na dané instrukci. V případě datových operací značí $operand_1$ vždy operand cílový a $operand_2$ operand zdrojový. Pozici v programové paměti označujeme řetězcem alfanumerických znaků, tzv. *label*, na který je možné odkazovat prostřednictvím větvících a podprogramových instrukcí.
- V oddílu 2.4 jsou vzájemně používány výrazy kontrolní logika a jednotka, případně řídící logika a jednotka. Význam všech uvedených výrazů je identitický. Dále mnemonická označení instrukcí, názvy stavů, bitových polí, zkratky aj. jsou vypisovány plnými názvy v anglickém jazyce, ze kterého byly původně odvozeny.
- Logické funkce jsou uváděny ve tvarech v souladu s notací matematické logiky. Namísto rovnítek, symbolů plus pro logický součet apod. jsou používány tradiční znaky tautologické ekvivalence, negace, konjunkce a disjunkce.
- Ve vzorových příkladech pro přehlednost vypisujeme pouze instrukční slova, paritní bity vynecháváme.
- V zadání práce můžeme nalézt spojení *koncepce syntetizovatelného mikroprocesoru*. Zde si dovolíme drobnou změnu použité terminologie a v dalším textu budeme používat označení *mikropočítač* namísto výrazu *mikroprocesor*. Přihlédneme-li k tomu, že součástí implementace jsou i integrované periférie a dále datová a programová paměť, zjístíme, že skutečně hovoříme o mikropočítači.

Kapitola 2

Návrh mikropočítače

Digitální počítač se obecně skládá ze tří fundamentálních částí a to z programové paměti, kontrolní jednotky a datové cesty. Datová cesta obsahuje všechny komponenty nezbytné pro provedení daných výpočtů a rovněž poskytuje úložiště pro výsledky z nich plynoucí. Operace prováděné na datové cestě jsou řízeny signály, které jsou generovány řídící jednotkou. Řídící jednotka je pak ve své podstatě deterministický konečný stavový automat, jehož chování je funkcí současného stavu a instrukčních vstupů z programové paměti (viz oddíl 2.4 Kontrolní jednotka). Právě takto je počítač realizován na nejvyšší z abstraktních úrovní, jak je patrné ze schématu níže.



Obr. 1: Zapojení všech tří konstrukčních bloků mikropočítače

Popis pinů

CLK – Clock

Hodinový vstup. Typické hodnoty se pohybují v řádech desítek MHz. Toto je jediný hodinový vstup mikropočítače, hodinový signál pro integrované periférie UART je od CLK odvozen (viz oddíl 3.1 UART).

RST – Reset

Reset mikropočítače. Po přivedení napájení bude zařízení v nedefinovaném stavu. Podržení resetovacího signálu ve vysoké úrovni po dobu alespoň jednoho hodinového cyklu zaručuje bezchybný rozběh stavového automatu kontrolní logiky. Reset je asynchronní a není tak přímo závislý na hodinovém signálu. Doporučení minimální délky pulzu vychází z požadavku na konečnou rychlost šíření signálů.

CA – Computer Alarm

Poplach počítače. Signalizuje přepnutí kontrolní logiky do chybového stavu a to v závislosti na jedné nebo více následujících událostí:

- Pokus o dekódování neplatné nebo rezervované instrukce.
- Kontrolní logika je v nedefinovaném nebo rezervovaném stavu.
- Selhání parity.
- Pokus o dělení nulou.
- Vypršení WDT časovače. Indikuje selhání uzavřeného cyklu stavového automatu kontrolní logiky.

Tento výstup typicky indikuje selhání zařízení. Pokud je tedy počítač například součástí složitějšího systému, nadřazené jednotky mohou vykonat patřičnou akci asociovanou s touto událostí, aktivovat zálohu či alespoň vyrozumět uživatele o situaci. Pro obnovení činnosti počítače je nutný reset.

IO – Input/Output

Obousměrný vstupně výstupní 12-bitový port. Slouží ke komunikaci s externími perifériemi. Port je konfigurovatelný pomocí dvojice instrukcí CHHP a CLHP (viz kapitola 4 Instrukční sada) a chová se v zásadě jako tří-stavový (*tri-state*) buffer . Na pin ve stavu vysoké impedance je možno nahlížet jako na rozpojený obvod a lze ho tak použít jako vstup. Externí vstupy IO portu jsou vedeny na vstup synchronizéru, který snímá vstupní data v souladu s náběžnou hranou hodinového signálu CLK. Synchronizér každého pinu obsauje tři *flip flop*y zapojené v konfiguraci posuvného registru. Takto je eliminován příjem nestabilního signálu.

URX – Uart Receiver

Vstupní vodič přijímacího modulu UARTu. Podobně jako v případě IO vstupů je i URX synchronizován. Nezaměňovat označení pinu URX (Uart receiver) s označením přijímacího registru URX (Uart Receiver Register).

UTX – Uart Transmitter

Výstupní vodič vysílacího modulu UARTu. Serializovaná data se objeví na výstupu i v případě, že je zařízení zapojeno do smyčky (UCR.LOP = 1). Nezaměňovat označení pinu UTX (Uart Transmitter) s označením vysílacího registru URX (Uart Transmitter Register).

Nyní můžeme přistoupit k podrobnému popisu a rozboru jednotlivých komponent mikropočítače jak je patrno z následujících oddílů.

2.1 Registry procesoru

V zájmu zachování jednoduchosti obsahuje procesor pouze dva univerzální registry A, B a univerzální paměť C (datová paměť). Registry A, B jsou určeny k rychlé manipulaci s okamžitými daty, libovolný počet dalších registrů, omezených pouze rozsáhlostí paměti, lze realizovat přímo v datové paměti. Překladač by tak například pro pohodlné programování mohl implementovat pseudo instrukce (složené z těch dedikovaných) právě k tomu určené.

```
0000  I I I 4       MACRO  MOVC(#reg, #immediate) IS
0000  I I I 4                      mov          a, #immediate
0002  RRR1                         sto          #reg
0002  RRR1          END MACRO
```

Tab. 1: Příklad pseudo instrukce[1]

Na jednoduchém příkladě výše je symbolicky naznačeno, jak by taková deklarace makra pseudoinstrukce mohla vypadat.

Další z registrů jsou nepřímo přístupný programový čítač PC a pro úplnost i čtyři interní, programově nepřístupné. Plné názvy registrů jsou zde uvedeny tak, aby odpovídaly svému mnemonickému vyjádření.

- **Accumulator A** – Slouží jako implicitní zdrojový a cílový operand pro některé z aritmeticko-logických a datových instrukcí. Registr akumuluje výsledky početních operací. Akumulátor je 12-bitový registr.
- **Auxiliary B** – Pomocný registr slouží jako implicitní zdrojový a cílový operand pro některé z aritmeticko-logických instrukcí. Aux B registr může být rovněž použit ve funkci ukazatele pro LOD a STO instrukce, což velmi usnadňuje kopírování daných úseků v datové paměti.
- **Memory C** – Univerzální datová paměť počítače adresovatelná instrukcemi LOD a STO. Komponenta představuje kromě datového úložiště rovněž rozhraní mezi procesorem a integrovanými periferiemi (viz oddíl 2.4.3 Datová paměť).
- **Address Register AR** – Tento registr slouží k adresaci dat z programové paměti. Hodnota obsažená v PC registru je v počátečním stavu S_0 zkopírována do AR a z paměti je tak možné přečíst instrukci.

[1] První čtyřčíslí představuje adresu dedikované instrukce v paměti, symboly I a R jsou okamžité hodnoty v závislosti na dodaných parametrech #reg a #immediate.

AR je 16-bitový registr, které je součástí kontrolní logiky a není tedy operandem žádné z makroinstrukcí.

- **Program Counter PC** – Obsahuje adresu instrukce, která má být v příštím cyklu vykonána. Registr může být manipulován pouze nepřímo pomocí kondicionálních instrukcí. Programový čítač je automaticky inkrementován na konci stavu S_1, tedy hned po nahrání instrukce z paměti. PC je 16-bitový registr.
- **Instruction Register IR** – Výstup z programové paměti je přímo zapisován do registru, jehož 4 dolní bity [3 – 0] tvoří právě IR. Instrukční registr tak obsahuje opkód a operand současné instrukce. Registr je pouze interní, je součástí kontrolní logiky.
- **Immediate Operand IMM** – Analogicky, IMM tvoří 12 horních bitů [15 – 4] výstupu z programové paměti. IMM registr kóduje okamžitou hodnotu (konstantu), což může být operand aritmeticko-logických a datových instrukcí či adresa programové paměti. Registr je pouze interní a je součástí kontrolní logiky a datové cesty.
- **Stack Pointer SP** – Zásobníkový ukazatel, ukazuje vždy aktuální návratou adresu.

15 12	11 4	0
	A	
	B	
	C	
AR (interní)		
PC		
		SP (interní)
IMM + IR (interní)		

Tab. 2: Přehled registrů procesoru

Kontrolní jednotka je popsána v oddílu 2.4, úplné znění všech instrukcí lze nalézt ve 4. kapitole.

2.2 Programová paměť

V programové paměti jsou zapsány sekvence instrukcí, podle které pak mikropočítač vykonává daný program. Paměť je pouze pro čtení (ROM) a z Obr. 1 lze vyčíst její začlenění do systému a jednotlivé porty. Zde je velmi důležité držet se určitých kódovacích konvencí a schémat a prakticky tak donutit syntetizér, aby paměť implementoval pomocí vlastních integrovaných RAM pamětí.

Ačkoliv je poměrně složité porovnávat mezi sebou jednotlivá FPGA, struktura desky a často i terminologie se liší v závislosti na výrobci, obecně lze říci, že každé FPGA bude obsahovat dva typy pamětí – DRAM (*Distributed RAM*) a BRAM (*Block RAM*).

Distribuovaná paměť je realizována pomocí velkého počtu LUT (*Look-up Table*), které jsou rozprostřeny (distribuovány) po celém FPGA. Bloková pamět je naopak skutečnou dedikovanou pamětí a její velikosti se pohybují v řádech stovek až tisíců kilobitů (kbit). Je patrné, že veliké paměti je vhodné ba přímo i

nutné implementovat právě pomocí blokových pamětí, neboť paralalení kombinace jednotlivých distribuovaných pamětí je prakticky synonymem pro problémové časování a synchronizaci.

Vzhledem k značné rozsáhlosti programové paměti je nutno adoptovat určité kódovací schéma, podle kterého syntetizér bezpečně pozná nejvhodnější typ paměti. Rozhodnutí a akce syntetizačních nástrojů jsou obecně založeny spíše na rozpoznávání určitých vzorů než na přímém zkoumání sémantického významu daného kódu. Ze stejného důvodu jednotliví výrobci poskytují ve své dokumentaci postupy, příklady nebo rovnou specifická makra a příkazy, kterémi lze požadovaného efektu dosáhnout.
V prvé řadě je bloková pamět synchronní, je tedy nutno poskytnout časovací signál CLK (CLOCK). Aktuální data na výstupu jsou tak k dispozici až na začátku (konci) hodinového cyklu, tedy během změny hodinového signálu. Říkáme, že výstup paměti je *registrován*. V případě distribuované paměti je výstup neregistrovaný a data je možno přečíst ihned, nezávisle na hodinovém signálu. Absence hodinového signálu tak automaticky vyústí v použití distribuované paměti.
Mikropočítač používá tzv. *single port*, jednoportovou paměť. Počet vstupních portů paměti je stejný jako počet výstupních portů, zde adresový vstup A (Address) a datový výstup DO (Data Output). Níže můžeme jednoduše porovnat schématické rozdíly mezi distribuovanou a blokovou pamětí.

```
ARCHITECTURE arch OF good_program_memory IS
TYPE memory_t IS ARRAY(65535 DOWNTO 0) OF STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL rom : memory_t;

BEGIN
    PROCESS(clk)
    BEGIN
        IF (rising_edge(clk)) THEN
            do <= rom(to_integer(unsigned(a)));
        END IF;
    END PROCESS;
END ARCHITECTURE;
```

Tab. 3: Výňatek z programové paměti realizované pomocí BRAM

Distribuovaná paměť může vypadat velmi jednoduše, viz příklad níže.

```
ARCHITECTURE arch OF bad_program_memory IS
TYPE memory_t IS ARRAY(65535 DOWNTO 0) OF STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL rom : memory_t;

BEGIN
    do <= rom(to_integer(unsigned(a)));
END ARCHITECTURE;
```

Tab. 4: Příklad chybně implementované paměti pomocí DRAM

Distribuovaná paměť je tedy vhodná spíše pro malé paměti, pro paměti větších rozměrů používáme blokové paměti.

Řídící jednotka počítače pracuje s 16-bitovým slovem, nicméně jak je patrné z příkladu v Tab. 3, data jsou v programové paměti uložena ve formě 18-bitových slov. Horní dva bity programového slova jsou tvořeny paritou a to následovně:

MSB		15	7	LSB
HP (High Parity)	LP (Low Parity)	UPW (Upper Program Word)	LPW (Lower Program Word)	

Tab. 5: Programové slovo

Parita je nejjednodušší typ kódu pro detekci chyb. Paritní bit určuje, zda dané slovo obsahuje sudý či lichý počet jedniček a vypočítá se prostým aplikováním XOR funkce bit po bitu zkoumaného slova. Paritní kód nedokáže odhalit všechny chyby a ty, které odhalí, také nedokáže opravit. V případě, že dojde k invertování dvou stejných bitů, nebude detekována žádná chyba. Tento problém je pochopitelně nejvíce znatelný u delších bitových posloupností. Z tohoto důvodu obsahuje programové slovo paritní bity dva, první (HP) pro horních 8 bitů (UPW) a druhý (LP) pro dolních 8 bitů (LPW) programového slova. Tím znatelně klesá pravděpodobnost, že dojde k převrácení dvou stejných bitů, které by připadly na jediný paritní bit.
Kontrolní jednotka přečte programové slovo z paměti a paritu instrukčního slova (viz oddíl 4.3) přepočítá. Pokud dojde k neshodě, přejde počítač do chybového režimu, tak jak je uvedeno v úvodu kapitoly.

Programová paměť dokáže pojmout 65536 instrukcí, celkem tedy 144 kB. Z pohledu řídící logiky je pameť adresována najednou (tzv. *flat addressing*) pomocí 16-ti adresových signálů. Uživatelský program však pracuje se segmentovanou pamětí, jak bude dále ukázáno ve 4. kapitole.

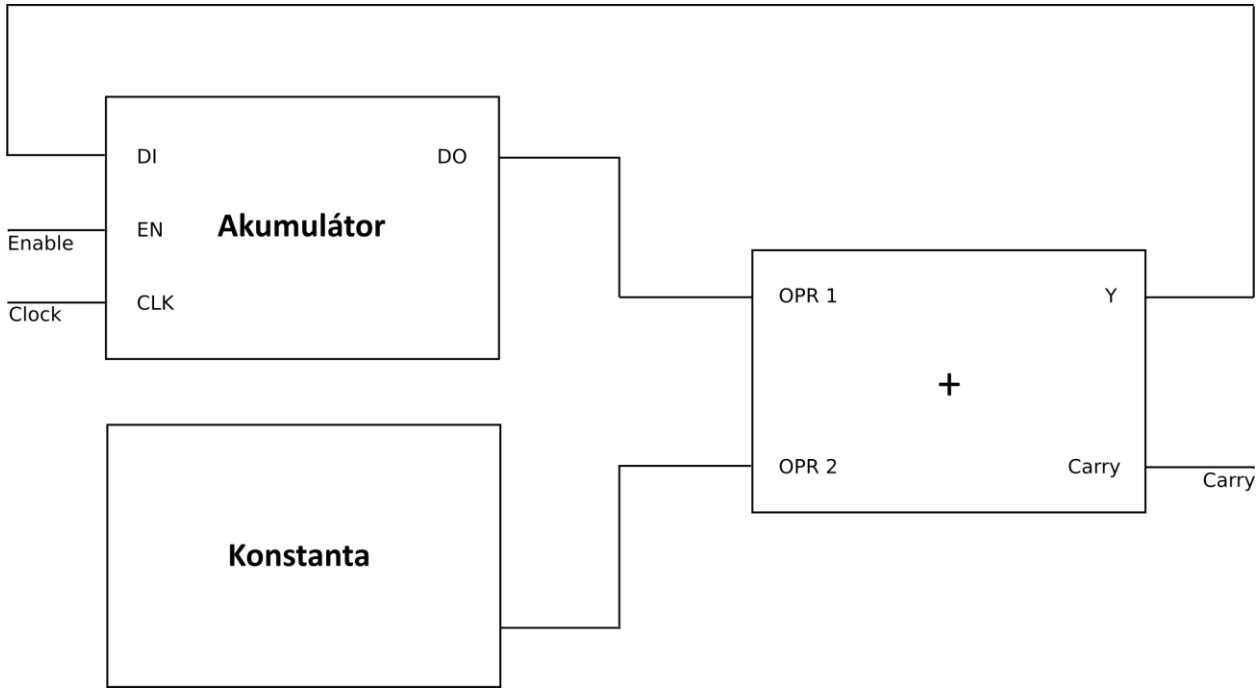
2.3 Datová cesta

V datové cestě (*z angl. datapath*) jsou soustředěny veškeré komponenty zodpovědné za prováděné operace mikropočítače. Datová cesta tak odráží skutečné výpočetní možnosti počítače. Zcela obecná datová cesta bude obsahovat množství kombinační logiky a paměťových elementů, které akumulují mezivýsledky početních operací a zároveň slouží jako vstupní operandy jednotlivých kombinačních bloků. Na zadaný problém tak nahlížíme abstraktně a výpočetní systém popisujeme z pohledu výměny dat mezi registry[2]. Hodnoty uložené v registrech se mohou měnit pouze při změně úrovně hodinového signálu, všechny operace prováděné na komponentách datové cesty tedy musí být dokončeny právě v tomto intervalu. Každou komponentu na cestě tak můžeme během jednoho hodinového cyklu použít pouze jednou. Toto je velmi výhodné, neboť nám to umožňuje přesně definovat instrukční sadu mikropočítače. Pokud bychom tedy například chtěli realizovat instrukci, která bude přičítat n-bitovou konstantu k obsahu n-bitového akumulátoru, je jisté, že datová cesta bude muset obsahovat

[2] RTL – Register Transfer Level

jednoduchou sčítačku s šířkou $n + 1$ bitů. Vstupem sčítačky bude akumulační registr a konstanta, výstup sčítačky pak bude směřován zpět do akumulátoru, tak jak je tomu na jednoduchém schématu níže.

-



Obr. 2: Příklad nejjednodušší datové cesty

Komponenty datové cesty jsou řízeny signály, které generuje řídící jednotka (viz oddíl 2.4). V uvedeném ilustračním příkladě máme jediný kontrolní signál EN (Enable), který řídí zápis výstupu sčítací logiky Y (Output) zpět do akumulačního registru. Prostým řízením kontrolních signálů tak můžeme vykonat libovolně složitý úkon, který může vyžadovat mnoho kroků a to vše prostřednictvím jediné komponenty. Pro n součtů tudíž nebude potřeba n sčítacích komponent, ale pouze jedna jediná, která bude použita právě n-krát.

Datová cesta mikropočítače je založena na stejném principu, je však rozsáhlejší a přirozeně obsahuje více komponent. Hlavní komponenty datové cesty počítače jsou aritmeticko-logická jednotka ALU, jednotka pro manipulaci s daty DTU a datová paměť. Tyto jsou také popsány v textu níže.

Blokové schéma použité datové cesty je k nalezení v závěru oddílu.

2.3.1 Aritmeticko logická jednotka

Aritmeticko logická jednotka[3], dále jen ALU, slučuje všechny výpočetní operace mikropočítače do jediné komponenty. Po aktivaci jednotky signálem CS (Chip Select) je možno zvolit požadovanou akci prostřednictvím signálu FS (Function Select), jehož možné hodnoty jsou uvedeny v tabulce níže.

FS. 2	FS. 1	FS. 0	Operace	Instrukce
FUNCTION SELECT				
0	0	0	Součet	ADD
0	0	1	Negace	NOT
0	1	0	Logický součin	AND
0	1	1	Logický součet	OR
1	0	0	Levý logicky posun	SLL
1	0	1	Pravý logický posun	SLR
1	1	0	Součin	MPY
1	1	1	Podíl	DIV

Tab. 6: Přehled aritmeticko logických operací mikropočítače

Přesný význam a kontrolní signály jednotlivých instrukcí, asociovaných s danou akcí jednotky, jsou k nahlédnutí v oddílu 4.4 čtvrté kapitoly textu.
Na prvním ALU vstupu OPR1 (Operand 1) je vždy připojen akumulátor, do kterého je rovněž směřován výstup Y (Output) jednotky. Na druhý vstup OPR2 (Operand 2) pak lze pomocí multiplexoru připojit pomocný registr B nebo registr okamžité hodnoty IMM. Druhým ALU výstupem je stavový signál FLG (Flags), který je vstupem stavového registru kontrolní jednotky.

MSB LSB

DIV	ZF	CF/REM

CF	FLG.0	Carry Flag.
REM	FLG.0	Remainder Flag.
ZF	FLG.1	Zero Flag.
DIV	FLG.2	Divide By Zero.

Tab. 7: Stavový Flag registr

Výklad jednotlivých polí je následující:

- **Carry Flag.** Pokud součet dvou dvanáctibitových operandů přesáhne hodnotu 12 bitů, CF bude nastavena na hodnotu 1. CF zde reprezentuje přestup o řád (*carry*) a právě tento bit je uložen ve formě stavové vlajky.

[3]Zkratky ALU – Arithmetical Logical Unit (početní operace) a dále DTU – Data Transfer Unit (výměna dat s registry, pamětí a okolními periferiemi).

- **Remainder Flag.** Zbytek po podílu obou operandů, kde $\mathrm{OPR1}$ je dělenec a $\mathrm{OPR2}$ dělitel, je nenulový. Čísla jsou nesoudělná, akumulátor bude obsahovat částečný podíl, CF bude nastavena na hodnotu 1 a samotný zbytek bude zahozen.
- **Zero Flag.** Výsledek ALU operace je nula, ZF bude nastavena na hodnotu 1. Stavovou vlajku ZF získáme negování logické součtu všech bitů výstupu $\mathrm{Y\ (Output)}$:

$$\mathrm{ZF} \rightleftharpoons \neg(\mathrm{Y}_0 \vee \mathrm{Y}_1 \ldots \mathrm{Y}_\mathrm{N}) \tag{1}$$

  kde $N = 11$ je šířka výstupu v bitech číslovaných od nuly.

- **Divide By Zero**. Operand $\mathrm{OPR2}$ je nulový. Při pokusu o dělení nulou bude DIV nastavena na hodnotu 1. Tato akce zároveň způsobí přechod mikropočítače do chybového stavu $\mathrm{CA\ (Computer\ Alarm)}$.

Podobně jako u součtu, tak i při násobení můžeme obržet výsledek přesahující datovou šířku 12 bitů. Obecně součin dvou 12 bitových čísel je 24 bitové číslo. Z tohoto důvodu je násobení provedeno ve dvou fázích. V první fázi instrukcí MPY nastaven signál $\mathrm{FS\ (Function\ Select)}$ na hodnotu $\mathrm{FS} = 6$ a další z ALU vstupů $\mathrm{MS\ (Multiply\ Select)}$ na hodnotu $\mathrm{MS} = 0$. Na výstup $\mathrm{Y\ (Output)}$ tak bude zapsáno spodních 12 bitů výsledku. V druhé fázi je instrukce MPY opakována, nyní však s $\mathrm{MS} = 1$. Takto obdržíme i zbývajících horních 12 bitů výsledné hodnoty. Ve skutečnosti ALU nepřepočítá stejný součin dvakrát (výsledek je uložen v interním 24 bitovém registru), což je ovšem nepodstatné, jelikož v jednom hodinovém cyklu může proběhnout stále pouze jedna z fází. Nezáleželo by tedy na tom, jestli ALU provede jeden *zbytečný* výpočet navíc, časová úspora je nulová.
Je nutno podotknout, že ačkoliv výsledek MPY instrukce může přesáhnout 12 bitů a tím přestoupit o řád, CF je v tomto případě irelevantní, neboť součin je 24 bitový (12-tý bit není *carry*). CF je tak platná opravdu pouze v případě součtové instrukce.
Signály CS, FS a MS jsou součástí kontrolní signálů, které jsou podrobněji popsány v oddílu 2.5, kontrolní logika.

Poznámka: Všimněme si, že implementovaná aritmeticko logická jednotka implicitně operuje pouze na číslech bez znaménka (*unsigned*). Konkrétní reprezentace kladných a záporných čísel může být řešena programově a to například pomocí jednotkového, resp. dvojkového doplňku (*one's complement, resp. two's complement*).

2.3.2 Datová jednotka

Úkolem datové jednotky je zápis konstant (okamžitých hodnot, *immediate*) do registrů a paměti a jejich vzájemná výměna. Datová jednotka rovněž zastává prostřední funkci mezi externími perifériemi a mikropočítačem. Jedním ze vstupu jednotky tudíž musí být i hodinový signál CLK, který synchronizuje zápis a čtení do IO registrů. IO port je dále rozbrán v oddílu 2.3.4, kde jsou v krátkosti zmíněny i ostaní podpůrné elementy datové cesty. Podobně jako v případě ALU je i zde konkrétní operace jednotky řízena kontrolním signálem FS (Function Select).

FS. 2	FS. 1	FS. 2	Operace	Instrukce
Function Select				
0	0	0	Přesun operandu	MOV
0	0	1	Zápis do IO	PIO IO
0	1	0	Čtení z IO	PIO A
0	1	1	Konfigurace portu	CHHP
1	0	0	Konfigurace portu	CLHP
1	0	1	Rezerva	Rezerva
1	1	0	Rezerva	Rezerva
1	1	1	Rezerva	Rezerva

Tab. 8: Přehled datových operací mikropočítače

Tímto způsobem je řízena operace s daty. Blokové schéma v závěru oddílu zahrnuje funkční bloky kombinační logiky jako je ALU (Arithmetical Logical Unit) a DTU (Data Transfer Logic), registry pro uložení okamžitých výsledků, datovou paměť a IO sběrnici pro komunikaci s okolními perifériemi.
DTU vpodstatě jenom zapisuje hodnotu na vstupu na výstup Y nebo IO. Vstupem může být okamžitá hodnota registru A, B, C a IMM (OPR) nebo hodnota přítomná na IO pinech (IO). Takto je možné vyměňovat data mezi registry a pamětí či do nich přímo zapisovat okamžité hodnoty.

2.3.3 Datová paměť

Datová paměť je velmi podobná té programové a platí tak pro ní všechny poznatky z příslušného oddílu (2.2). Opět, vzhledem k její velikosti musí být realizována pomocí dedikovaných BRAM pamětí. Na rozdíl od programové paměti je však možné její obsah libovolně měnit pomocí uživatelských instrukcí – takovou paměť pak nazýváme pamětí RAM (*Random Access Memory*). Abychom mohli do paměti zapisovat, je nutno zavést další dva vstupy a to DI (Data Input) a ENC (Enable C).
K adresování libovolné buňky slouží 11-bitový signál A (Address). Vzhledem k tomu, že adresování datové paměti je lineární (namísto segmentovaného u programové paměti), šířka jedné buňky je 12 bitů, můžeme říci, že celkový paměťový prostor čítá 3072 bytů. Pomocí signálu ENC pak volíme, zda chceme obsah dané buňky přečíst či ji naopak přepsat hodnotou novou. Datový vstup je registrován, data zapsána do paměti tak budou k dispozici až v příštím hodinovém cyklu.

Kromě univerzálního úložiště slouží datová paměť jako rozhraní mezi mikropočítačem a integrovanými periferiemi. Celkem šest paměťových lokací v rozmezí od 0x7F8 do 0x7FF je tak rezervováno pro příslušné registry daných periférií. Jejich krátkých přehled lze najít v tabulce níže.

Lokace	Směr	Mnemonika	Registr
Paměťově mapované periférie			
7F8	Čtení	UART CR	Control Register
7F9	Zápis	TIM CMPH	Compare High
7FA	Zápis	TIM CMPM	Compare Med
7FB	Zápis	TIM CMPL	Compare Low
7FC	Čtení	TIM STS	Timer Status
7FD	Zápis	UART TX	Uart Trasmitter
Paměťově mapované periférie (pokrač.)			
7FE	Čtení	UART RX	Uart Receiver
7FF	Zápis	UART CR	Control Register

Tab. 9: Rezervované paměťové pozice

Podrobný popis jednotlivých paměťově mapovaných registrů lze najít v třetí kapitole o integrovaných periféiích.

Jak již bylo dříve zmíněno, datová paměť musí být realizována pomocí BRAM. Toto však také automaticky znamená restrikci v počtu vstupních a výstupních portů paměti. Ačkoliv se paměťový signál může jevit jako jednodimenzionální pole známé z vysokoúrovňových programovacích jazyků, nemůžeme k dedikovaným paměťovým pozicím (registrům) přistupovat prostým zapsáním indexu v závislosti na volené adrese. Takovýto zápis by znemožnil syntetizačními nástroji použít blokové paměti a došlo by tak k utilizaci obrovského počtu LUT bloků (DRAM). Místo toho je nutno dané registry číst paralelně s regulérní datovou pamětí a konečný výstup řídit multiplexorem, jako je tomu na příkladu níže.

```
ARCHITECTURE arch of memory_c IS
TYPE memory_t IS ARRAY(2047 TO 0) of STD_LOGIC_VECTOR(11 DOWNTO 0);
SIGNAL ram : memory_t;
SIGNAL read_io : STD_LOGIC_VECTOR(1 DOWNTO 0);

...

BEGIN
WITH read_io SELECT do <= ram(a_reg)  WHEN “00”,
                          sts_reg   WHEN “01”,
                          uart_rx   WHEN “10”,
                          uart_crr WHEN “11”,
                          ram(a_reg)   WHEN OTHERS;
```

```
PROCESS(clk)
    BEGIN
        IF(rising_edge(clk)) THEN
            CASE a_reg IS
                WHEN 16#7FC# =>
                    read_io <= "01";
                WHEN 16#7FE# =>
                    read_io <= "10";
                WHEN 16#7F8# =>
                    read_io <= "11";
                WHEN OTHERS =>
                    read_io <= "00";
            END CASE;
        ....

        END IF;
    END PROCESS;
END ARCHITECTURE;
```

Tab. 10: Výňatek z datové paměti

Multiplexor přepíná výstup paměti v závislosti na řídícím signálu read_io. Pokud je tedy zvolena dedikovaná paměťová adresa, výstup paměti bude ukazovat hodnotu na výstupu patřičného periferního registru. Tímto způsobem zůstane zachován původní počet portů paměti – jeden vstup a jeden výstup.

2.3.4 Ostatní komponenty

Datová cesta dále obsahuje podpůrné prvky jako multiplexory a demultiplexory, které přepojují jednolivé vstupy a výstupy mezi komponentami v závislosti na generovaných řídících signálech, jak bude vyloženo v následujícím oddílu.
Na schématu v závěru oddílu můžeme nalézt čtyři multiplexory, dva identické zdrojové, jeden cílový a adresový, a jeden demultiplexor. Oba zdrojové multiplexory připojují registry a datovou paměť (operandy) ke vstupu komponent ALU a DTU. Vstupy obou komponent jsou tedy v každém okamžiku totožné a odpovídající komponentu je nutno volit prostřednictvím signálu CS (Chip Select), který je také jedním ze signálů kontrolních. Na schématickém obrázku si lze také povšimnout, že vstup CS demultiplexoru je trvale připojen na úroveň log. 1 a je ve skutečnosti CS signálem pouze řízen (viz Tab. 12). Cílový multiplexor řídí zpětný zápis do registrů a adresový multiplexer přepojuje zdroj adresy, kterým může být buď pomocný registr B nebo adresa okamžitá.

Vraťme se ještě ke komponentě DTU. Jak již bylo řečeno, DTU obsahuje jeden obousměrný (*bi-directional*) IO port. Protože tří stavová (*tri-state*) logika není v digitálním systému realizovatelná[4], je nutné vždy danému pinu přiřadit směr vstupu nebo výstupu. Samotný IO port je tedy řešen prostřednictvím tří signálů a to IOINPT (IO Input), IOUPT (IO Output) a IOCFG (IO Configuration).

[4] Neexistuje *skutečně* obousměrný port a digitální systém pracuje pouze s binárními hodnotami.

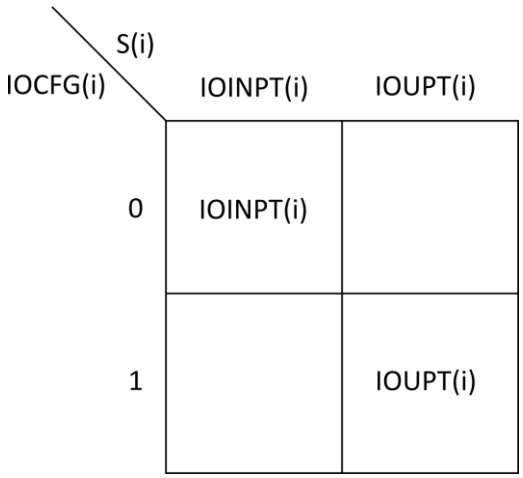
Všechny tři signály jsou stejné délky jako výsledný signál IO (Input and Output), který je také vyveden na samostatných IO pinech mikropočítače. Označení jednotlivých signálů napovídá, že IOINPT má směr vstupu a IOUPT směr výstupu. Konfigurační signál IOCFG pak přepojuje vstupní a výstupní signály na IO.

Aby mohl být pin IO signálu použit jako vstup, nesmí být řízen žádným interním signálem, tzn. musí být ve stavu vysoké impedance (*floating*). V tuto chvíli je možné tento pin zatížít (řídit) vnějším zdrojem, například periférií. Logická hodnota pinu může být následně přečtena mikropočítačem.

Vstup	Výstup	Směr	Tri − state
SIGNÁLY OBOUSMĚRNÉHO IO PORTU *S(i)*			
IOINPT(i)	IOUPT(i)	IOCFG(i)	IO(i)
		0	IOINPT(i)
		1	IOUPT(i)

Tab. 11: IO port multiplexor

Princip činnosti IO portu (multiplexoru) je shrnut v kompaktně zapsané pravdivostní tabulce výše. Index i značí individuální bit a pin signálu, respektive portu. Tento zápis působí značně neintuitivně a proto výslednou funkci pro úplnost zapíšeme z jednoduché Karnaughovy mapy.



Obr. 3: IO port multiplexor

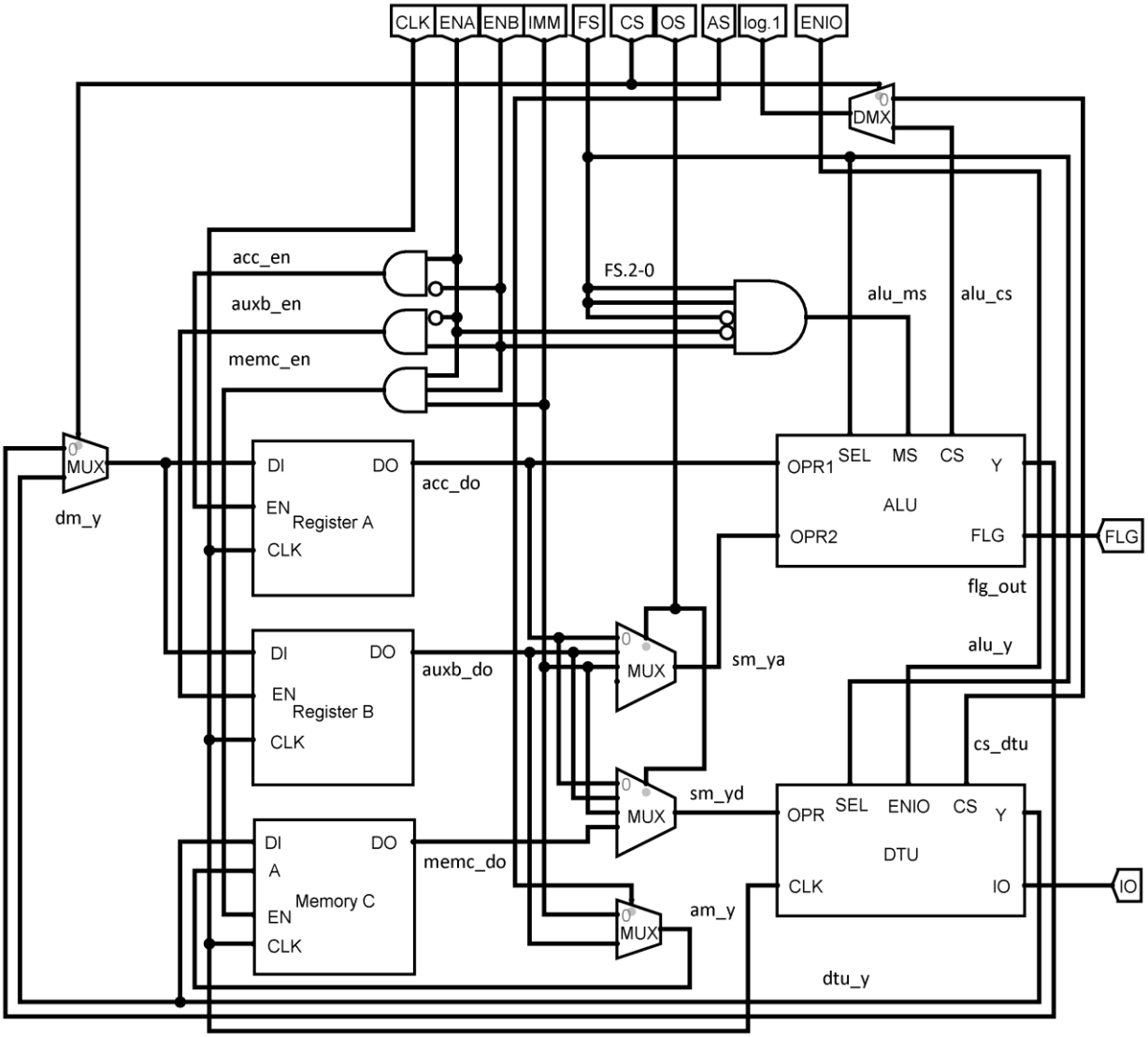
Platí tedy, že

$$\mathrm{IO}(i) \bowtie \left(\neg \mathrm{IOCFG}(i) \wedge \mathrm{IOUPT}(i)\right) \vee \left(\mathrm{IOCFG}(i) \wedge \mathrm{IOINPT}(i)\right) \tag{2}$$

což není nic jiného, než logická funkce 2-1 multiplexoru. Směr IO portu je volen instrukcemi CHHP a CLHP, jak bude uvedeno v odílu 4.4 dále. Multiplexor je velmi jednoduché zařízení, nicméně krátké pozastavení je určitě namístě, neboť struktura HDL jazyků svádí k prosté deklaraci IO signálu klíčovým slovem INOUT. Jak nyní vidíme, toto v syntetizovatelném popisu není možné. K přímé deklaraci INOUT se uchylujeme až v nejsvrchnější entitě a to z důvodů výše popsaných.
Význam všech signálů, kterými jsou multiplexory a demultiplexory řízeny, je uveden v oddílu níže.

2.3.5 Schéma datové cesty

Níže je úplné schéma datové cesty mikropočítače. Vstupy datové cesty jsou vyobrazeny v horní části schématu, tedy CLK (Clock), ENA (Enable Accumulator), ENB(Enable Auxiliary), IMM (Immediate), FS (Function Select), OS (Operand Select), AS (Address Select) a ENIO (Enable IO). Výstupy datové cesty jsou pak vyvedeny v pravé části obrázku a to FLG (Flags) a IO (Input And Output).



Obr. 4: Datová cesta mikropočítače

Kromě vstupních kontrolních a výstupních signálů obsahuje datová cesta ještě interní signály, které jednotlivé komponenty propojují[5]. Následující tabulka je všechny přehledně shrnuje. Tyto jsou rovněž ve schématu vyznačeny.

[5] Značení interních signálů je totožné se značením užitých ve zdrojových kódech.

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

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

Tab. 12: Výčet interních signálů datové cesty

Poznámka: Z úsporných důvodů je ve schématu zobrazen signál FS jako jediný vodič. Aby mohl být spočten MS (Multiply Select) signál, je nutné mezi sebou logicky vynásobit jednotlivé bity FS signálu. Jednotlivké vstupy AND brány v pořadí od horního k dolnímu vstupu tedy jsou $\mathrm{FS.2}, \mathrm{FS.1}, \mathrm{FS.0}, \mathrm{ENA}$ a ENB.

2.4 Kontrolní jednotka

Signály potřebné k vlastní obsluze datové cesty jsou generovány řídící jednotkou. Řídící jednotku je výhodné popsat pomocí konečného stavového automatu (*Finite-state machine, FSM*). V případě deterministického automatu se systém může nacházet v jednom okamžiku pouze v jediném, přesně definovaném stavu, přičemž změna stavu je možná pouze na základě množiny vstupů a současného stavu. Formálně můžeme přechodovou funkci δ stroje zapsat jako

$$\delta : S \times \Sigma \to S \tag{3}$$

kde $S \neq \emptyset$ je konečná množina strojových stavů a $\Sigma \neq \emptyset$ je konečná množina vstupů. Výstup je funkcí vstupů a současného stavu, takovýto stroj bývá označován jako Mealyho stroj (*Mealy machine*).

Následuje úplný zápis všech použitých stavů kontrolní jednotky. Stavy, které nejsou v tabulce uvedeny, jsou tímto považovány za rezervované a teoreticky není možné do takovéhoto stavu vstoupit. Pokud bychom totiž excitační tabulku překreslili do stavového diagramu, což není nic jiného než orientovaný graf, jehož vrcholy značí jednotlivé stavy stroje a hrany funkčně závislé přechody, zjistíme, že do rezervovaných stavů jednoduše nevede žádná hrana. Deterministický stroj se tudíž v takovém stavu nemůže nikdy nacházet. Platí však, že reálná zařízení se působením okolních vlivů mohou do nedefinovaného stavu dostat. Toto je obzvláště pravděpodobné, pokud je stroj provozován v podmínkách, na které není dimenzován (zejména vysokoenergetické záření či excesivní teploty). Z těchto důvodů je výhodné *nahradit* každý nevyužitý stav některým ze stavů již definovaným, nejlépe resetovacím či chybovým. V našem případě byl zvolen stav $S_{31\ (1Fh)}$ (Computer Alarm, viz úvod kapitoly 2).

Stavový diagram kontrolní logiky v této práci z praktických důvodů uváděť nebudeme.

CURRENT STATE	NEXT STATE	INPUT					OUTPUT						
0 – 3F	0 – 3F	OP	OPR	EXT	MOD	EXE	CS	FS	OS	AS	ENA	ENB	END
TRANSITION FROM S_0 STATE – FETCH AND WRITE BACK STAGES													
00	01	X[6]	X	X	X	X	X	X	X	X	0	0	0
02	03	X	X	X	X	1	0	0	1	X	1	0	0
02	00	X	X	X	X	0	X	X	X	X	0	0	0
03	00	X	X	X	X	X	X	X	X	X	0	0	0
04	03	X	X	X	X	X	0	0	3	X	1	0	0
05	0C	X	X	X	X	1	0	1	1	X	0	1	0
05	00	X	X	X	X	0	X	X	X	X	0	0	0
06	03	X	X	X	X	1	0	2	1	X	1	0	0
06	00	X	X	X	X	0	X	X	X	X	0	0	0
07	03	X	X	X	X	X	0	2	3	X	1	0	0
08	03	X	X	X	X	1	0	3	1	X	1	0	0
08	00	X	X	X	X	0	X	X	X	X	0	0	0
09	03	X	X	X	X	X	0	3	3	X	1	0	0
0A	03	X	X	X	X	X	1	0	3	X	1	0	0
0B	0C	X	X	X	X	X	1	0	3	X	0	1	0
0C	00	X	X	X	X	X	X	X	X	X	0	0	0
0D	03	X	X	X	X	1	1	0	0	X	1	0	0
0D	00	X	X	X	X	0	X	X	X	X	0	0	0
0E	0C	X	X	X	X	1	1	0	1	X	0	1	0
0E	00	X	X	X	X	0	X	X	X	X	0	0	0
0F	00	X	X	X	X	1	1	1	0	X	0	0	1
0F	00	X	X	X	X	0	X	X	X	X	0	0	0
10	03	X	X	X	X	1	1	2	1	X	1	0	0
10	00	X	X	X	X	0	X	X	X	X	0	0	0
11	00	X	X	X	X	X	X	X	X	X	0	0	0
12	00	X	X	X	X	X	X	X	X	X	0	0	0
13	00	X	X	X	X	X	1	0	0	0	1	1	0
14	03	X	X	X	X	X	1	0	2	0	1	0	0
15	03	X	X	X	X	1	0	4	3	X	1	0	0
15	00	X	X	X	X	0	X	X	X	X	0	0	0
16	03	X	X	X	X	1	0	5	3	X	1	0	0
16	2A	X	X	X	X	0	1	3	3	X	0	0	1
16	2A	X	X	X	X	0	1	4	3	X	0	0	1
17	20	X	X	X	X	1	0	6	1	X	0	0	0
17	00	X	X	X	X	0	X	X	X	X	0	0	0
18	03	X	X	X	X	1	0	7	1	X	1	0	0
18	00	X	X	X	X	0	X	X	X	X	0	0	0
19	00	X	X	X	X	1	1	0	0	1	1	1	0
19	00	X	X	X	X	X	X	X	X	X	0	0	0
1A	03	X	X	X	X	1	1	0	2	1	1	0	0

[6] X – *Don't care*, signál je v dané interpretaci irelevantní.

CURRENT STATE	NEXT STATE	INPUT					OUTPUT						
0 – 3F	0 – 3F	OP	OPR	EXT	MOD	EXE	CS	FS	OS	AS	ENA	ENB	END
TRANSITION FROM S_0 STATE – FETCH AND WRITE BACK STAGES (cont.)													
1A	00	X	X	X	X	0	X	X	X	X	0	0	0
1B	1E	X	X	X	X	X	X	X	X	X	0	0	0
1C	1E	X	X	X	X	1	X	X	X	X	0	0	0
1C	00	X	X	X	X	0	X	X	X	X	0	0	0
1D	26	X	X	X	X	X	0	0	3	X	1	0	0
1E	00	X	X	X	X	X	X	X	X	X	0	0	0
1F	1F	X	X	X	X	X	X	X	X	X	X	X	X
20	21	X	X	X	X	X	0	6	1	X	1	0	0
21	00	X	X	X	X	X	0	6	1	X	0	1	0
22	27	X	X	X	X	1	0	0	3	X	0	0	0
22	23	X	X	X	X	0	X	X	X	X	0	0	0
23	24	X	X	X	X	1	1	0	0	0	1	1	0
23	24	X	X	X	X	0	X	X	X	X	0	0	0
24	00	X	X	X	X	X	X	X	X	X	0	0	0
26	22	X	X	X	X	X	X	X	X	X	1	0	0
27	23	X	X	X	X	X	X	X	X	X	1	0	0
28	1E	X	X	X	X	X	X	X	X	X	0	0	0
29	01	X	X	X	X	X	X	X	X	X	0	0	0
2A	00	X	X	X	X	X	X	X	X	X	0	0	0
TRANSITION FROM S_1 STATE – DECODING STAGE													
01	02	0	0	0	X	X	0	0	1	X	0	0	0
01	04	0	1	X	X	X	0	0	3	X	0	0	0
01	05	1	1	X	X	X	0	1	1	X	0	0	0
01	06	2	0	0	X	X	0	2	1	X	0	0	0
01	07	2	1	X	X	X	0	2	3	X	0	0	0
01	08	3	0	0	X	X	0	3	1	X	0	0	0
01	09	3	1	X	X	X	0	3	3	X	0	0	0
01	0A	4	0	0	X	X	1	0	3	X	0	0	0
01	0B	4	1	X	X	X	1	0	3	X	0	0	0
01	0D	5	0	0	X	X	1	0	0	X	0	0	0
01	0E	5	1	X	X	X	1	0	1	X	0	0	0
01	0F	6	0	0	X	X	1	X	0	X	0	0	0
01	10	6	1	X	X	X	1	X	1	X	0	0	0
01	11	7	0	0	X	X	X	X	X	X	0	0	0
01	12	7	1	X	X	X	X	X	X	X	0	0	0
01	13	1	0	0	1	X	1	0	0	0	1	1	0
01	14	1	0	0	0	X	1	0	2	0	0	0	0
01	15	0	0	1	0	X	0	4	3	X	0	0	0
01	16	0	0	1	1	X	0	5	3	X	0	0	0
01	17	2	0	1	0	X	0	6	1	X	0	0	0
01	18	2	0	1	1	X	0	7	1	X	0	0	0
01	19	5	0	1	1	X	1	0	0	1	1	1	0
01	1A	5	0	1	0	X	1	0	2	1	0	0	0
01	1B	3	0	1	0	X	X	X	X	X	0	0	0

CURRENT STATE	NEXT STATE	INPUT					OUTPUT						
0 – 3F	0 – 3F	OP	OPR	EXT	MOD	EXE	CS	FS	OS	AS	ENA	ENB	END
TRANSITION FROM S_1 STATE – DECODING STAGE (cont.)													
01	1C	3	0	1	1	X	X	X	X	X	0	0	0
01	1D	6	0	1	X	X	0	0	3	X	0	0	0
01	2A[7]	0	0	1	0	X	X	X	X	X	0	0	0

Tab. 13: Excitační tabulka kontrolní logiky

Je možné si povšimnout, že některé stavy nepřiřazují kontrolním signálům žádné hodnoty. Tyto stavy záměrně neovlivňují komponenty datové cesty a namísto toho spouští některou z interních akcí řídící jednotky. Příkladem je například CALL instrukce, která manipuluje s adresovým zásobníkem přímo v řídící jednotce. Pro větší přehlednost jsou názvy stavů s číselným označením[8] shrnuty v tabulce níže.

S_{HEX}	S_{DEC}	Název	Poznámka
Numerické a mnemonické označení			
00	00	Fetch0	Výchozí stav
01	01	Fetch1	Dekódování
02	02	ADD B	
03	03	Acc Write-Back	
04	04	ADD IMM	
05	05	NOT B	
06	06	AND B	
07	07	AND IMM	
08	08	OR B	
09	09	OR IMM	
0A	10	MOV IMM	
0B	11	MOV IMM	
0C	12	Aux Write-Back	
0D	13	MOV R	
0E	14	MOV R	
0F	15	PIO IO	
10	16	PIO A	
11	17	B	Interní
12	18	BEQ	Interní
13	19	STO IMM	
14	20	LOD IMM	
15	21	SLL/CMS	CMS interní

S_{HEX}	S_{DEC}	Název	Poznámka
Numerické a mnemonické označení (pokrač.)			
16	22	SLR/CHHP/CLHP	
17	23	MPY	
18	24	DIV	
19	25	STO B	
1A	26	LOD B	
1B	27	CALL	Interní
1C	28	RET	Interní
1D	29	DAB	Sekvence
1E	30	Stack OFF	Interní
1F	31	Comp. Alarm	Interní
20	32	MPY Write-Back	MS = 0
21	33	MPY Write-Back	MS = 1
22	34	DAB Decrement	
23	35	DAB Write-Back	
24	36	DAB Branch	
25	37	RSVD	Rezerva
26	38	DAB Write-Back	
27	39	DAB Write-Back	
28	40	Read Stack	Interní
29	41	Wait	
2A	42	PIO Write-Back	
X	X	(Prázdné)	

Tab. 14: Přehled všech FSM stavů

[7] Instrukce CMS bude ve stavu S_1 vždy rozpoznána jako SLL a dvojice instrukcí CHHP a CLHP jako SLR, neboť z úsporných důvodů sdílejí stejné dekódovací bity OP, OPR, EXT a MOD. Vlastní dekódování tak proběhne až ve SLL (SLR) stavu.

[8] Číselné označení stavů je pouze orientační a odpovídá generickému značení ve zdrojových kódech. Skutečné kódování stavů je provedeno až při syntéze; syntetizér často volí *one-hot* kódování.

Kontrolní logika tedy generuje sedm řídících signálů, které ovládají činnosti jednotlivých komponent mikropočítače. Každý ze signálů tak lze chápat jako elementární akci, jejichž *posloupnost* pak představuje patřičnou makroinstrukci. Popis signálů je uveden v následující tabulce; paralelní posloupnost jednotlivých signálů, které korespondují k jednotlivým instrukcím jsou pak uvedeny v samostatné kapitole. Ve výčtu jsou pro pořádek uvedeny i výstupy z registru okamžité hodnoty IMM a MS (Multiply Select) ačkoliv sami o sobě nepatří do kategorie řídících signálů.

<table>
<tr><td>CS.0</td><td>Chip Select. Prostřednictvím toho signálu vybírá demultiplexor ALU nebo DTU komponentu. Pokud $\mathrm{CS} = 0$, komponenta ALU je aktivována, v opačném případě $(\mathrm{CS} = 1)$ je aktivována komponenta DTU. Pro aktivovanou komponentu se pak stává relevatním signál FS. Výstup demultiplexoru je vždy log. 1 pro aktivovanou komponentu a log.0 pro deaktivovanou komponentu. Pokud je komponenta deaktivovaná, její výstup Y zůstává nezměněn. Signál CS rovněž ovládá cílový multiplexor, pomocí kterého je veden zpětný zápis do registrů.</td></tr>
<tr><td>OS.0-1</td><td>Operand Select. Tento signál ovládá multiplexor který připojuje datové výstupy registrů A,B, C a IMM na zdrojové vstupy komponent ALU a DTU. Možné hodnoty kterých může OS.0-1 nabývat je uveden v tabulce níže.
<table>
<tr><th>OS1</th><th>OS0/CS[9]</th><th>Zdroj − ALU/DTU</th><th>Zdroj − A/B</th></tr>
<tr><td>0</td><td>0</td><td>Registr A</td><td>ALU</td></tr>
<tr><td>0</td><td>1</td><td>Registr B</td><td>DTU</td></tr>
<tr><td>1</td><td>0</td><td>Registr C</td><td>NC[10]</td></tr>
<tr><td>1</td><td>1</td><td>Registr IMM</td><td>NC</td></tr>
</table></td></tr>
<tr><td>FS.1-0</td><td>Function Select. Tento signál vybírá specifickou akci kterou by komponenta měla provést. Cílová komponenta musí mít CS log. 1, jinak je tento signál ignorován a výstup dané komponenty se nemění. Platné hodnoty, jakých může signál nabývat, jsou uvedeny v tabulce níže pro každou z komponent zvlášť.
<table>
<tr><th>FS2</th><th>FS1</th><th>FS0</th><th>ALU akce</th><th>DTU akce</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>ADD</td><td>MOV</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>NOT</td><td>PIO IO</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>AND</td><td>PIO A</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>OR</td><td>CHHP</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>SLL</td><td>CHLP</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>SLR</td><td>Rezerva</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>MPY</td><td>Rezerva</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>DIV</td><td>Rezerva</td></tr>
</table></td></tr>
<tr><td>AS.0</td><td>Address Select. Tento signál ovládá adresový multiplexor, jehož výstup je směřován na adresový port A datové paměti. Multiplexor tak vybírá mezi adresový vstupy, kterými mohou být registry okamžité hodnoty IMM $(\mathrm{AS} = 0)$nebo registr B $(\mathrm{AS} = 1)$.</td></tr>
</table>

[9] Operand Select OS.0 pro zdroj ALU/DTU, resp. CS pro zdroj A/B. Výstup cílového multiplexoru je vyveden na datové vstupy obou registrů A a B. CS signál pak určuje, zda je na tento výstup zapsán výsledek ALU nebo DTU operace. Přirozeně, pokud CS aktivoval např. ALU komponentu, tak data očekávaná na vstupu patřičného cílového registru musí pocházet právě z téže komponenty, tj. ALU.

[10] *Not Connected*, nepřipojeno.

<table>
<tr><td>IMM.3-0</td><td>Immediate Output. Obsahuje výstup z registru okamžité hodnoty IMM, který je přiveden na vstup multiplexorů pro komponenty ALU a DTU. Pokud $\mathrm{OS} = 3$, objeví se tato hodnota IMM registru na OPR vstupech obou komponent. IMM není kontrolní signál.</td></tr>
<tr><td>ENA.0[11]</td><td>Enable Accumulator. Ovládá zápis do registru A. Pokud $\mathrm{ENA} = 1$ obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna. Pouze jeden ze signálů ENA a ENB může být log. 1 v každém okamžiku.</td></tr>
<tr><td>ENB.0</td><td>Enable B. Ovládá zápis do registru B. Pokud $\mathrm{ENB} = 1$ obsah registru bude při příštím hodinovém cyklu přepsán, v opačném případě zůstane hodnota v registru nezměněna. Pouze jeden ze signálů ENA a ENB může být log. 1 v každém okamžiku.</td></tr>
<tr><td>END.0</td><td>Enable IO. Hlavní EN signál, který ovládá individuální EN signály vstupních, výstupních a konfiguračních IO registrů. Alias signálu je ENIO. Kontrolované signály jsou shrnuty v tabulce níže.

<table>
<tr><th>END</th><th>FS</th><th>Funkce</th><th>Signál/Reg</th><th>Instrukce</th></tr>
<tr><td>1</td><td>1</td><td>ENOUT</td><td>IOUPT</td><td>PIO IO</td></tr>
<tr><td>1</td><td>2</td><td>ENIN</td><td>IOINPT</td><td>PIO A</td></tr>
<tr><td>1</td><td>3</td><td>ENHHP</td><td>IOCFG</td><td>CHHP</td></tr>
<tr><td>1</td><td>4</td><td>ENLHP</td><td>IOCFG</td><td>CLHP</td></tr>
</table>

<table>
<tr><td>ENOUT</td><td>Enable Output.</td><td>$\mathrm{ENOUT} \bowtie \mathrm{END} \wedge \neg \mathrm{FS}.2 \wedge \neg \mathrm{FS}.1 \wedge \mathrm{FS}.0$</td></tr>
<tr><td>ENIN</td><td>Enable Input.</td><td>$\mathrm{ENOUT} \bowtie \mathrm{END} \wedge \neg \mathrm{FS}.2 \wedge \mathrm{FS}.1 \wedge \neg \mathrm{FS}.0$</td></tr>
<tr><td>ENHHP</td><td>Enable High Half Port.</td><td>$\mathrm{ENOUT} \bowtie \mathrm{END} \wedge \neg \mathrm{FS}.2 \wedge \mathrm{FS}.1 \wedge \mathrm{FS}.0$</td></tr>
<tr><td>ENLHP</td><td>Enable Low Half Port.</td><td>$\mathrm{ENOUT} \bowtie \mathrm{END} \wedge \mathrm{FS}.2 \wedge \neg \mathrm{FS}.1 \wedge \neg \mathrm{FS}.0$</td></tr>
</table>

Poznámka: Pravdivostní tabulka je zde zapsána velmi úsporně. Signál (funkce) nabývá hodnoty log.1 pouze pro uvedené hodnoty END a FS. Jinde je funkce nulová.</td></tr>
<tr><td>MS.0</td><td>Multiply Select. Pro hodnoty $\mathrm{MS} = 0$ bude zapsáno spodních 12 bitů a pro $\mathrm{MS} = 1$ horních 12 bitů ALU výstupu. Používáno MPY instrukcí. MS není kontrolní signál.</td></tr>
</table>

Tab. 15: Přehled kontrolních signálů

Signály ENA a ENB nabývají vždy komplementárních hodnot nebo jsou právě nulové. Pokud by mělo platit ENA = ENB = 1, došlo by k simultánímu zápisu do obou registrů A a B, což by znehodnotilo data v nich obsažená. Z důvodu úspory počtu řídících signálů využívá datová paměť této zmíněné restrikce.

Instrukce STO po dekódování položí ENA = ENB = 1, čímž signalizuje paměťový zápis. Kombinace ENA, ENB a hodnota IMM[MSB] tak produkuje signál ENC, který je přiveden na vstup C komponenty a ovládá její zápis. Zároveň však stále musí platit, že nedojde k simultánímu zápisu do obou registrů A a B, což ani není záměrem instrukce STO. Tuto jednoduchou úvahu lze zapsat formou pravdivostní tabulky níže.

[11] Signály EN mohou nabývat log. 1 pouze při WB fázi.

IMM[MSB]	ENA	ENB	ENA′	ENB′	ENC
VSTUPNÍ SIGNÁLY			**VÝSTUPNÍ SIGNÁLY**		
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	1	0	0
1	1	1	0	0	1

Tab. 16: Signály ENA, ENB a ENC

Signály ENA a ENB jsou generovány přímo řídící logikou, zatímco ENA′ a ENB′ jsou pak připojeny k Enable vstupům jednotlivých registrů. Tyto tři funkce, ve tvaru CNF[12], tedy jsou

$$\mathrm{ENA}' \bowtie \mathrm{ENA} \wedge \neg\mathrm{ENB} \tag{4}$$
$$\mathrm{ENB}' \bowtie \neg\mathrm{ENA} \wedge \mathrm{ENB} \tag{5}$$
$$\mathrm{ENC} \bowtie \mathrm{IMM[MSB]} \wedge \mathrm{ENA} \wedge \mathrm{ENB} \tag{6}$$

Je patrné, že k zápisu do datové paměti C dojde pouze tehdy, pokud jsou signály IMM[MSB], ENA a ENB v logické úrovni log. 1. Ačkoliv ENA = ENB = 1, k zápisu do registrů A a B nedojde, neboť na Enable vstupu bude log. 0. Konečně pokud bude IMM[MSB] v log. 0, pro dvojici instrukcí STO/LOD, zachovává si ENA a ENB svůj původní význam a na Enable vstupu obou registrů bude ENA' = ENA a ENB' = ENB.

V podobném duchu je z kontrolních signálů odvozen i signál MS (Multiply Select). Jakých hodnot signál nabývá je pro nás přirozeně zajímavé pouze v případě, kdy je prováděno násobení MPY. Zkrácený zápis ve formě pravdivostní tabulky je k nalezení níže.

FS. 2	FS. 1	FS. 0	ENA	ENB	MS
VSTUPNÍ SIGNÁLY					**VÝSTUP**
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0
X	X	X	X	X	X

Tab. 17: Signál MS

Z pravdivostní tabulky můžeme zapsat funkci ve tvar CNF, tedy

$$\mathrm{MS} \bowtie \mathrm{FS.2} \wedge \mathrm{FS.1} \wedge \neg\mathrm{FS.0} \wedge \neg\mathrm{ENA} \wedge \mathrm{ENB} \tag{7}$$

[12] CNF – Konjunktivní normálová forma (Conjuctive Normal Form)

Stav S_{20H} tudíž zapíše zpět do registru A dolních 12 bitů součinu $(\mathrm{MS} = 0)$, zbylých horních 12 bitů bude zapsáno do registru B v následujícím stavu S_{21H} $(\mathrm{MS} = 1)$.

Vstupem kontrolní jednotky je instrukční slovo (viz oddíl 4.3) a signál EXE (Execute), který určuje, zda bude daná instrukce vykonána či přeskočena. Signál EXE tudíž umožňuje podmíněné vykonávání instrukcí v závislosti na stavových vlajkách FLG registru. Signál opět popíšeme pomocí pravdivostní tabulky.

YE	ZE	Y	Z	EXE
VSTUPNÍ SIGNÁLY				**VÝSTUP**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1

YE	ZE	Y	Z	EXE
VSTUPNÍ SIGNÁLY (pokrač.)				**VÝSTUP**
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Tab. 18: Signál EXE

Je patrné, že EXE je funkcí signálů (dekódovacích bitů) YE (Carry Enable), ZE (Zero Enable) a Y (Carry Flag CF), Z (Zero Flag ZF). Platí tedy (DNF[13])

$$\mathrm{EXE} \bowtie (\neg \mathrm{YE} \wedge \neg \mathrm{ZE}) \vee (\neg \mathrm{YE} \wedge \mathrm{Z}) \vee (\mathrm{ZE} \wedge \mathrm{Z}) \vee (\mathrm{YE} \wedge \mathrm{Y}) \qquad (8)$$

Instrukce pro kterou platí $\mathrm{EXE} = 0$ bude přeskočena a kontrolní logika se vrátí do výchozího stavu S_0.

Na první pohled by se mohlo zdát, že má kontrolní jednotka poměrně veliký počet stavů. Ve skutečnosti však stroj prochází pouze pěti fázemi.

- **(1) Vyčtení instrukce.** Ve výchozím stavu S_0 jsou všechny Enable signály $\mathrm{ENA}, \mathrm{ENB}, \mathrm{ENC}$ a END položeny úrovni log. 0. Tímto je ukončen jakýkoliv probíhající zpětný zápis do registrů či datové paměti. Pokud poslední vykonaná instrukce byla aritmeticko-logická, je načten obsah FLG registru. Dále je zkopírován obsah PC do AR registru. Data z programové paměti budou nicméně k dispozici až v příštím hodinovém cyklu. Konečně je resetován signál CL (Computer Lockup) WDT časovače (viz oddíl 2.5.6). Aktivováním resetovacího signálu RST přejde mikropočítač právě do tohoto stavu.
- **(2) Čekání na dokončení paměťové operace.** Výstup programové paměti je registrován a je tudíž o jeden hodinový cyklus zpožděn. Stav S_{29H} tak plní pouze funkci časové prodlevy.
- **(3) Dekódování.** Ve stavu S_1 dochází k spočtení parity a EXE signálu a hlavně k dekódování vyčtené instrukce. Právě v dekódovací fázi dochází ke generování řídících signálů datové cesty. Zároveň je inkrementován PC registr, který nyní ukazuje na adresu následující instrukce.

[13] DNF – Disjunktivní normálová forma (Disjunctive Normal Form)

- **(4) Vykonání instrukce.** Po dekódování následuje daný počet instrukčních stavů. Tato fáze může u jednoduchých instrukcí typu ADD trvat jediný hodinový cyklus (stav), složitější instrukce však může zabrat podstatně více hodinových cyklů (typicky DAB). Platí, že výstupy komponent datové cesty musí být před koncem instrukční fáze bezpečně stabilizovány.
- **(5) Zpětný zápis.** Po ustálení výstupů datové cesty následuje zpětný zápis (Write-Back) výsledků do registrů, datové paměti či vstupně výstupních portů a návrat do výchozího stavu S_0. Instrukce, které neoperují na paměťových komponentách datové cesty fázi zpětného zápisu přirozeně postrádají a rovnou se tak vrací do výchozího stavu.

Posloupnost všech těchto fází pak tvoří jeden *instrukční cyklus*. Proces dekódování instrukce je zachycen na procesním diagramu, který je vzhledem ke svému značnému rozsahu zařazen mezi přílohy na úplném konci textu.

2.4.1 Zásobník

Zásobník (*stack*) je paměť určená výhradně k vkládání a obnovování návratových adres volacích instrukcí. Data v zásobníku jsou manipulována dvojicí operací a to *push* a *pop*. V závislosti na směru, kterým zásobník roste (nahoru nebo dolů) je snižován či zvyšován zásobníkový ukazatel SP (Stack Pointer), který odkazuje na aktuální pozici v zásobníkové struktuře. Operace *push* vloží do zásobníku nové slovo a sníží (*zvýší*) SP. Analogicky *pop* odebere ze zásobníku slovo a zvýší (*sníží*) SP. Takovéto schéma je tradičně označováno jako LIFO (*Last In, Last Out*). Vložíme-li (*push*) tedy do zásobníku posloupnost A, B, C, D obdržíme data po opětovném vyčtení (*pop*) v pořadí D, C, B, A.
Zásobník implementovaný kontrolní jednotkou obsahuje 16 paměťových pozic o šířce instrukčního slova (16 bitů). Tento zásobník je interní a lze s ním nepřímo nakládat pouze pomocí CALL a RET instrukcí. Funkce datové struktury je tedy striktně omezena pro účely podprogramových volání. Přesný princip činnosti zásobníku je patrný z výňatku níže.

```
PROCESS(clk, rst)
BEGIN
    IF(rst = '1') THEN
        sp <= 15;
    ELSIF(rising_edge(clk)) THEN
        IF(EN = '1') THEN
            IF(SEL = '1') THEN
                stack(sp) <= di;
                IF(sp /= 0) THEN
                    sp <= sp - 1;
                ELSE
                    sp <= 15;
                END IF;
                do <= di;
            ELSIF(sel = '0') THEN
                IF(sp /= 15) THEN
                    sp <= sp + 1;
```

```
                    do <= stack(sp + 1);
                ELSE
                    sp <= 0;
                    do <= stack(0);
                END IF;
            END IF;
        END IF;
    END IF;
END PROCESS;
```

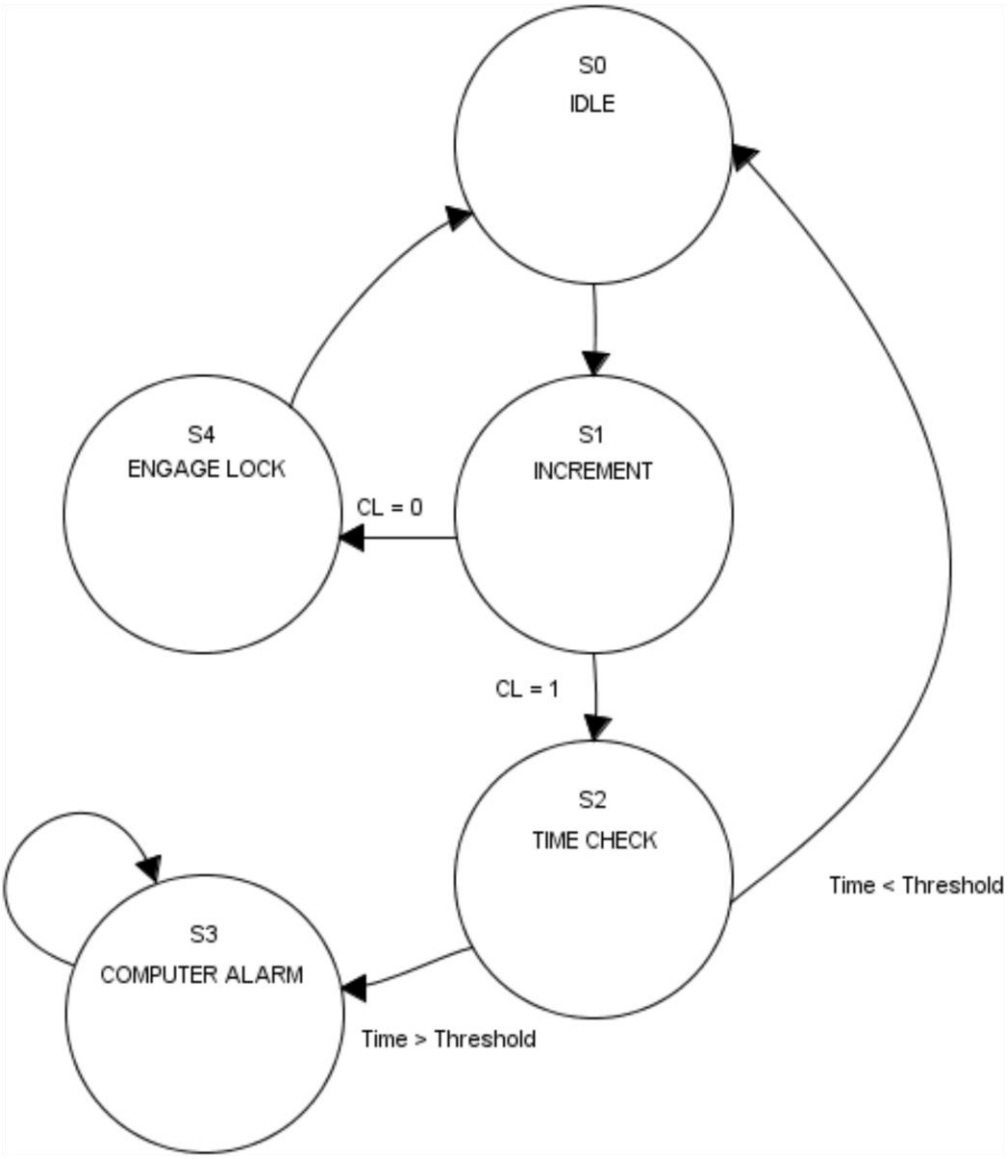
Tab. 19: Zásobníková struktura

Zásobník roste směrem dolů. Signál $\mathrm{SEL} = 1$ (Select) značí vklad datového slova do struktury (*push*), zatímco $\mathrm{SEL} = 0$ vyčtení datového slova ze struktury (*pop*). V případě, že je překročena kapacita zásobníku, nastává *overrun* (*push* nad rámec kapacity) nebo *overflow* (*pop* nad rámec kapacity). Z výše uvedeného jednoduše vyplývá, že maximální hloubka je 16 úrovní volání podprogramu. Případné pochybení je tedy na straně vlastního programu a stavy *overrun* a *overflow* jsou kontrolní jednotkou tiše ignorovány.

2.4.2 Watchdog

Watchdog (doslova *hlídací pes*), zkráceně WDT (*Watchdog Timer*), je jednoduchý čítač, který počítá do předem stanovené hodnoty. Tento čítač je periodicky resetováván počítačem. Pokud z jakéhokoliv důvodu dojde k selhání počítače, čítač nebude resetován, vyprší a vygeneruje chybový signál. Na základě chybového signálu může nadřazená jednotka vykonat patřičnou akci a pokud možno navrátit systém zpět do definovaného stavu a obnovit jeho činnost. Watchdog je ideální do autonomních, nepřetržitě běžících systémů nebo pro aplikace, kde je údržba vyžadující lidský faktor nemožná (typicky vesmírné instalace).
Zde implementovaný watchdog je resetován ve výchozím stavu S_0 vynulováním signálu CL (Computer Lockup). Hranice vypršení čítače je rovna počtu hodinových cyklů nejdelší instrukce, kterou je v našem případě DAB. Takto je zaručeno, že se stavová logika mikropočítače nedostane vinou okolních vlivů do nekonečné smyčky.

Funkci WDT časovače můžeme tradičně popsat pomocí konečného stavového automatu, viz níže. Zde se, narozdíl od kontrolní logiky, omezíme na stavový diagram. Alternativní metody popisu, tedy stavové a excitační tabulky, jsou zcela ekvivalentní.



Obr. 5: Stavový diagram WDT časovače

Ve stavu S_1 dochází k inkrementaci čítače. Pokud je signál CL (Computer Lockup) stále v úrovni log. 1, znamená to, že mikropočítač právě vykonává jednu z fází definovaných výše. Pokud je pak ve stavu S_2 zjištěno, že instrukční cykus nebyl dokončen včas, je vygenerován chybový signál CA (Computer Alarm), který zastaví činnost počítače přechodem kontrolní logiky do stavu S_{1FH}. V opačném případě se časovač navrací do výchozího stavu S_0 a celý proces se opakuje.

Kapitola 3

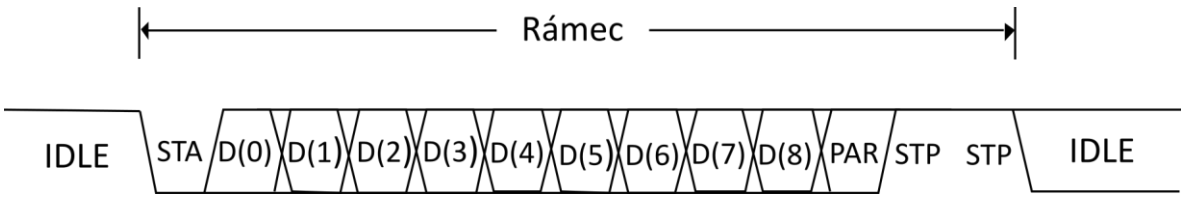
Integrované periférie

3.1 UART

UART (*Universal Asynchronous Receiver and Transmitter*) je zařízení, které převádí paralelní tok dat na sériový a zpět. UART se používá k sériové komunikaci a je na místě zdůraznit, že pracuje pouze s logickými úrovněmi signálů. Napěťové úrovně, význam a definice fyzicky přenášených signálů podléhají patřičným standardům, z nichž jmenujme alespoň několik nejpoužívanějších a to hlavně RS-232 a RS-485. To nám umožňuje soustředit se přímo na vlastní funkční návrh modulu a nikoliv na detaily fyzické implementace. UART obsahuje přijímací a vysílací obvody, které jsou popsané v jednotlivých oddílech níže.

3.1.1 Přijímač

Přijímací modul UARTu funguje principiálně jako posuvný registr (*deserializer*), do kterého jsou pomocí vzorkovače sériově ukládána příchozí data. Po naplnění registru je možné data přečíst paralelně (výstup posuvného registru je paralelní). Sériová data se shromažďují do tzv. rámců. Struktura datového rámce je zobrazená níže.



IDLE	Klidový stav	Linka UARTu je v klidovém stavu[14] vždy v úrovni log.1.
STA	Start bit	Záčátek rámce je uvozován vždy log.0 po dobu trvání jednoho symbolu.
D(i)	Data bit	Užitečný náklad rámce. Standardně se přenáší 6,7 nebo 8 datových bitů.
PAR	Paritní bit	Parita může být sudá, lichá nebo fixní. Paritní bit lze také úplně vynechat.
STP	Stop bit	Stop bit signalizuje konec rámce. Vždy log.1, přenáší se 1,1.5 nebo 2 stop bity.

Obr. 6: UART rámec

[14] Vysoká logická úroveň je zde volena z historických důvodů. Vyšší hodnoty napětí klidového stavu linky dovolují okamžitě odhalit přerušené vedení.

Protokol neumožňuje žádnou výměnu informací o použitých komunikačních kmitočtech a nastavení formátu rámce. Jednotlivé stanice tudíž musí manuálně zvolit identitickou symbolovou rychlost a délku rámce – počet datových, paritních a stop bitů. Nejpoužívanější symbolová rychlost je 9600 baud/s. Aby byl přenos úspěšný, musí cílová stanice vzorkovat jednotlivé bity s periodou rovnou převrácené hodnotě symbolové rychlosti. Pokud je tedy zvolena například symbolová rychlost 9600 baud/s a jeden symbol odpovídá jednomu bitu, vzorkovač by měl pracovat na frekvenci 9,6 kHz. V praxi se však využívá převzorkování (*oversampling*) a to nejčastěji 16-ti násobek symbolové rychlosti (kmitočtu). Následující postup vzorkování je volným překladem z [5].

1. Přijímač průběžně naslouchá na lince a vyčkává na začátek start bitu. Start bit (začátek rámce) je rozpoznán jako pokles logické úrovně linky na log.0. Právě v tuto chvíli spouští přijímací stanice vzorkovací čítač (viz dále).
2. Jakmile vzorkovací čítač napočítá do hodnoty 7, pak okamžitá hodnota přijímaného signálu je nyní přesně ve středu start bitu. Čítač je resetován a začíná počítat opět od nuly.
3. Po 15 odpočítaných pulzech se signál posunul právě o jeden bit a vzorkovač je nyní přesně ve středu prvního datového bitu. Tento bit je přečten a uložen do posuvného registru. Čítač je resetován a začíná počítat opět od nuly.
4. Pro přijetí zbylých datových bitů je nutno krok 3 opakovat $3N - 1$ krát, kde N je počet datových bitů.
5. Pokud protokol používá paritní bit, je krok 3 ještě jednou zopakován.
6. Konečně je krok 3 zopakován $3M$ krát, kde M je počet stop bitů. Přečtením všech stop bitů je přijímání rámce ukončeno.

Obecně platí, že celý HDL návrh by měl být vždy řízen pouze jediným hodinovým signálem. Zavedením více hodinových domén může docházet k synchronizačním problémům mezi jednotlivými signály a dalším zbytečným komplikacím. Vzorkovací *hodinový* signál je tedy nutno odvodit od CLK (Clock) signálu a to pomocí čítače, kterému se říká (*pre*)*scaler*. Funkce takového obvodu je patrná z HDL popisu níže.

```
PROCESS(clk, rst)
BEGIN
    IF(rst = '1') THEN
        scaler_counter <= preset_scaler;
        ovr_clk <= '0';
    ELSIF(rising_edge(clk)) THEN
        IF(scaler_counter = 0) THEN
            scaler_counter <= preset_scaler;
            ovr_clk <= '1';
        ELSE
            scaler_counter <=  scaler_counter – 1;
            ovr_clk <= '0';
        END IF;
    END IF;
END PROCESS;
```

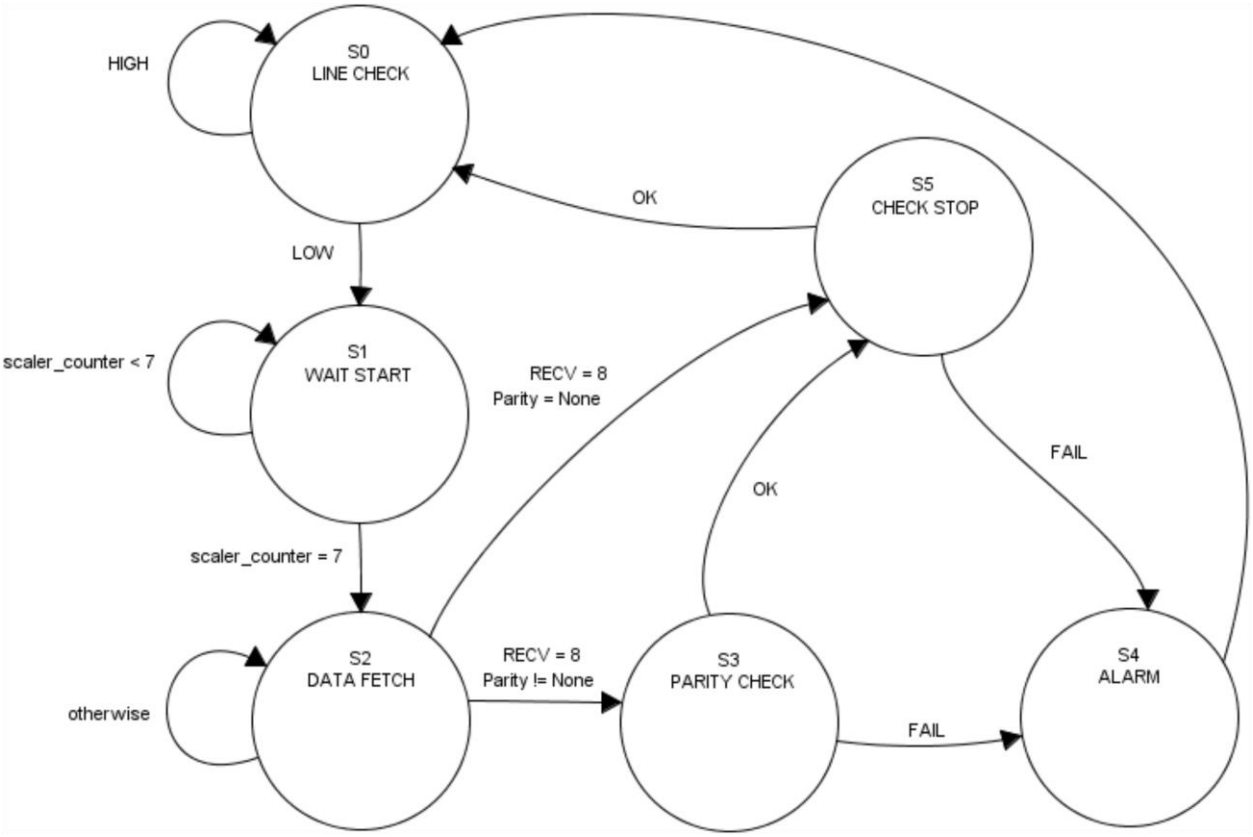
Tab. 20: Výňatek z přijímacího UART modulu, oversampler

Signál preset_scaler je funkcí CLK (Clock) kmitočtu a uživatelem volené symbolové rychlosti. Platí,

$$\text{preset\_scaler} = \frac{\text{CLK kmitočet}}{16 \cdot \text{symbolová rychlost}} \frac{[\text{s}^{-1}]}{[\text{s}^{-1}]} \tag{9}$$

kde jediným proměným parametrem je symbolová rychlost; hlavní hodinový kmitočet je generován pomocí oscilátoru a nelze ho tak měnit. Symbolovou rychlost je možno nastavit v kontrolním registru UARTU, viz dále.
Tímto způsobem je stále možné řídit přijímací modul UARTu hlavním hodinových signálem. Vzorkovací *hodinový* signál ovr_clk (Oversampled Clock) tak funguje pouze jako CE (Clock Enable). Po dobu plnění čítače je tento CE v úrovni log. 0, krátký pulz o délce převrácené hodnoty CLK kmitočtu je vygenerován až po naplnění čítače. Právě v tuto chvíli je odebrán vzorek příchozího signálu.

V souladu s postupem uvedeným na začátku oddílu je přijímač realizován pomocí konečného stavového automatu. Opět se zde omezíme na stavový diagram.



Obr. 7: Stavový diagram přijímacího stupně UART modulu

Pro přehlednost ještě uvedeme symbolický výpis akcí jednotlivých stavů přijímače a jejich vliv na kontrolní registr UCR (Uart Control Register).

S_N	S_{N+1}	Vstup	UCR	Poznámka
NUMERICKÉ OZNAČENÍ		**ZJEDNODUŠENÝ ZÁPIS VSTUPŮ A VÝSTUPŮ**		
0	0	Linka v klidovém stavu	$\mathrm{BSY} = 0$	
0	1	Změna stavu linky	$\mathrm{BSY} = 1$	Detekována hrana start bitu.
1	1	$\mathrm{scaler\_counter} < 7$	-	Navzorkováno méně než 8/16.
1	2	$\mathrm{scaler\_counter} = 7$	-	Prostřední hodnota start bitu.
2	2	$\mathrm{RECV} < 8$	-	Přijato méně než 8 data bitů.
2	3	$\mathrm{RECV} = 8, \mathrm{Parity} \neq \mathrm{None}$	-	V příštím stavu bude přijat paritní bit.
2	5	$\mathrm{RECV} = 8, \mathrm{Parity} = \mathrm{None}$	-	V příštím stavu bude přijat stop bit.
3	5	OK	$\mathrm{UA} = 0$	-
3	4	FAIL	-	Chyba parity (*Parity Error*).
4	0	-	$\mathrm{UA} = 1$	Příčina selhání zde není rozlišena.
5	0	OK	$\mathrm{RXF} = 1$	-
5	4	FAIL	-	Přijat chybný rámec (*Framing Error)*.

Tab. 21: Přehled stavů přijímacího UART modulu

Data jsou platná až po dokončení přenosu. Po přijetí celého slova bude platit $\mathrm{RXF} = 1$ (Receiver Full). Jelikož systém nemá žádnou paměť (typicky FIFO[15] zásobník), je nutné přijatá data načítat průběžně, neboť jsou na výstupu ponechána pouze po dobu rovnou podílu délky rámce v bitech a symbolové rychlosti – to znamená do naplnění posuvného registru dalším slovem. Přirozeně čím je rychlost přenosu vyšší, tím kratší dobu má mikropočítač na přečtení slova z přijímače. Pro symbolovou rychlost 9600 baud/s zůstavají data na výstupu po dobu přibližně 1 ms.

Po přečtení přijatého slova z výstupního registru UART_RX je nutné vlajku RXF resetovat. Pokud není výstup včas načten do paměti mikropočítače, jsou data ztracena (přepsána). Tento stav označujeme jako *Overrun* a je indikován pokusem o zapsání naplněného posuvného registru na výstup modulu pokud zároveň platí $\mathrm{RXF} = 1$.

Korektní zpracování příchozího rámce programem je ukázáno na příkladu níže.

```
0000  7F81                    ORG 0
0000  7F81          ; Input: None, Output: A data read
0000  7F81          .uart_read:   lod     #UART_CR
0001  000D                        mov     b, a
0002  200A                        and     #0x200
0003  804F                        beq     .uart_read      ; RXF = 0
0004  0005                        mov     a, b
0005  1FFA                        and     #0x1FF          ; Clear RXF
0006  FFF1                        sto     #UART_CR
0007  7FE1                        lod     #UART_RX
0008  8013                        ret                     ; Return
```

Tab. 22: Doporučený postup pro příjem

[15] FIFO – First In First Out

Poznámka: Po úspěšném dokončení operace bude RXF = 1. Pokud došlo při přenosu k chybě, hodnota RXF zůstane nezměna a UA = 1. Pokud byla vlajka uživatelem resetována, nenabyde tedy hodnoty log. 1. Takto je zaručeno, že přijatá data jsou vždy platná.

3.1.2 Vysílač

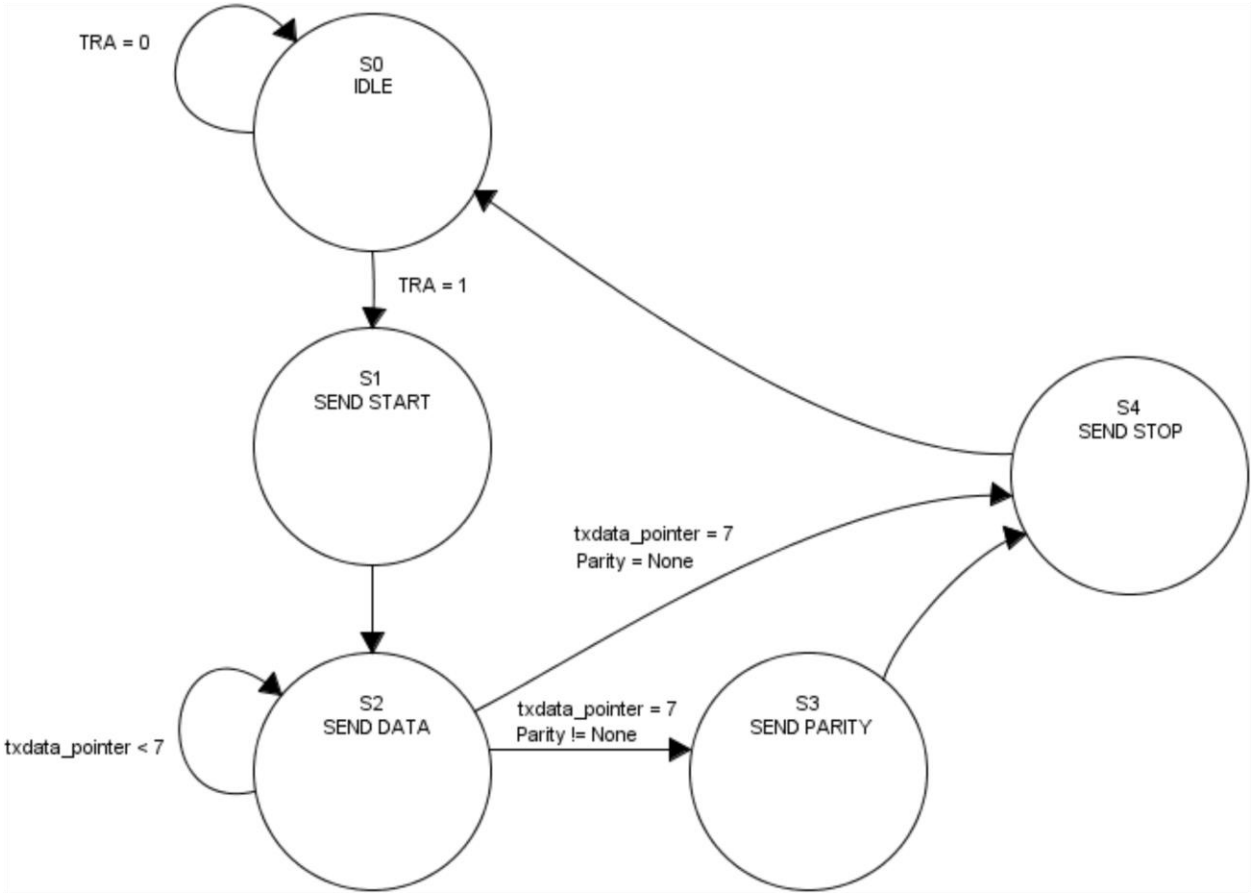
Vysílací modul UARTu je znatelně jednodušší než modul přijímací popisovaný v oddílu výše. Celý vysílač je v podstatě pouze posuvný registr (*serializer*), který posouvá bity na sériový výstup rychlostí, která odpovídá právě rychlosti symbolové. Opět je však důležité dodržet schéma jednotné hodinové domény. Vysílací *hodinový* takt tedy odvodíme od hlavního hodinového signálu CLK (Clock), podobně jako v případě přijímací části. Princip je zcela totožný, pouze nyní je generovaný signál 16-ti násobkem vzorkovacího *hodinového* signálu ovr_clock(Oversampled Clock).

```
PROCESS(clk, rst)
BEGIN
    IF(rst = '1') THEN
        baud_counter <= preset_scaler * 16;
        baud_clk <= '0';
    ELSIF(rising_edge(clk)) THEN
        if(baud_counter = 0) THEN
            baud_counter <= preset_scaler * 16;
            baud_clk <= '1';
        ELSE
            baud_counter <= baud_counter – 1;
            baud_clk <= '0';
        END IF;
    END IF;
END PROCESS;
```

Tab. 23: Výňatek z vysílacího UART modulu, baud generátor

Signál baud_clk(Baud Clock) zde opět funguje jako CE(Clock Enable). Pokud vynásobíme rovnici pro preset_scaler faktorem 16, dostaneme opět původní škálovací hodnotu pro kmitočet symbolové rychlosti. Identicky, po dobu plnění čítače baud_counter bude CE v úrovni log. 0, krátký pulz je vygenerován až v okamžiku naplnění čítače. Právě v tomto okamžiku je vyčten (posunut) bit z posuvného registru a zapsán na sériový výstup.

Proces vysílání je zachycen na stavovém diagramu níže.



Obr. 8: Stavový diagramu vysílacího stupně UART modulu

Pro přehlednost ještě uvedeme symbolický výpis akcí jednotlivých stavů přijímače a jejich vliv na kontrolní registr UCR (Uart Control Register).

S_N	S_{N+1}	Vstup	UCR	Poznámka
NUMERICKÉ OZNAČENÍ		**ZJEDNODUŠENÝ ZÁPIS VSTUPŮ A VÝSTUPŮ**		
0	0	Vysílač připraven, $\mathrm{TRA} = 0$	$\mathrm{BSY} = 0$	
0	1	Vysílač aktivován, $\mathrm{TRA} = 1$	$\mathrm{BSY} = 1$	
1	2	-	-	
2	2	$\mathrm{txdata} < 8$	-	Serializováno méně než 8 data bitů.
2	3	$\mathrm{txdata} = 8, \mathrm{Parity} \neq \mathrm{None}$	-	V příštím stavu bude vyslána parita.
2	4	$\mathrm{txdata} = 8, \mathrm{Parity} = \mathrm{None}$	-	V příštím stavu bude vyslán stop bit.
4	0	-	$\mathrm{BSY} = 0$ $\mathrm{TRA} = 0$	-

Tab. 24: Přehled stavů vysílacího UART modulu

Jak je vidět, stavová vysílací logika je téměř zrcadlovým obrazem logiky přijímací. Nejprve je naplněn vstupní registr UART TX daty, které chceme vyslat. Nastavením kontrolního bitu TRA (Transmit) jsou tato data zvalidována a přenos může začít. V následujících stavech pak budou vysílány start, datové, paritní a stop bity a to vždy když $CE = 1$. Konec přenosu rámce je signalizován resetováním stavových vlajek TRA a BSY. Pro úplnost je nastavena i TXE (Transmitter Empty).
Na příkladu níže je ukázáno praktické ovládání vysílacího modulu UARTu.

```
0000  0000                    ORG 0
0000  0000          Input: A data to send, Output: None
0000  FFB3    .uart_write:    sto     #UART_TX
0002  7F81    .line_bsy:      lod     #UART_CR
0004  001A                    and     #1
0006  002F                    beq     .line_rdy
0008  8047                    b       .line_bsy
000A  0803    .line_rdy:      or      #0x80       ; TRA = 1
000B  0000                    sto     #UART_CR
000C  7F81    .uart_tra:      lod     #UART_CR
000E  080A                    and     #0x100      ; TXE = 0 ?
0010  8031                    beq     .uart_tra   ; Block
0012  8013                    ret                 ; Return
```

Tab. 25: Doporučený postup pro vysílání

Návrat z procedury bude vykonán až po dokončení přenosu. Stavová vlajka TXE (Transmitter Empty) je pouhou negací TRA (Transmit), tedy platí $TXE = 1$ když $TRA = 0$ a $TRA = 1$ když $TXE = 0$. Mohlo by se tak zdát, že obě vlajky jsou zcela ekvivalentní a zbytečné. Kontrolní bit TRA je však pouze pro zápis, TXE pro čtení. Z praktických důvodů je rovněž přehlednější kontrola nulového bitu a ušetří se tím jedna instrukce. Jelikož je procedura blokovací[16], lze po jejím skončení okamžitě vysílat další data.

[16] Program zde předpokládá bezchybnou funkci zařízení. V realné aplikaci by bylo nutno doplnit proceduru o časové vypršení operace (*timeout*). V případě selhání modulu by jinak mikropočítač skončil v nekonečné (programové) smyčce.

3.1.3 Kontrolní a jiné registry

UCR : UART CONTROL REGISTER **Zápis: 0x7FF** **Čtení: 0x7F8**

MSB LSB

RSVD	LOP	RXF	TXE	TRA	RST	PR1	PR0	BD1	BD0	UA	BSY

BSY	UCR.00	R	**Busy.** Indikuje data na lince, modul právě přijímá, případně vysílá. Automaticky nastaveno a resetováno přijímačem či vysílačem. Použití linky je možné pouze v případě neobsazené linky, tj. BSY = 0.
UA	UCR.01	RW	**Uart alarm.** Byla detekována chyba rámce, *overrun*, neplatný FSM stav či chyba parity. Stavovou vlajku lze zápisem pouze resetovat. UA musí být resetováno programem.
BD0	UCR.02	W	**Baud rate.** Spodní bit symbolové rychlosti, viz Tab. 27 níže.
BD1	UCR.03	W	**Baud rate.** Horní bit symbolové rychlosti, viz Tab. 27 níže.
PR0	UCR.04	W	**Parity.** Spodní bit nastavení parity, viz Tab. 27 níže.
PR1	UCR.05	W	**Parity.** Horní bit nastavení parity, viz Tab. 27 níže.
RST	UCR.06	W	**Reset.** Nutné pro inicializaci zařízení. Je zvolen výchozí FSM stav (IDLE) a po dobu držení resetu je veškerá komunikace inhibitována. Resetovací procedura je asynchronní a je nutno ji manuálně dokončit opětným zápisem do UCR s RST = 0.
TRA	UCR.07	W	**Transmit.** Validuje data zapsaná do UTX registru a zahajuje vysílání. Automaticky resetováno vysílačem.
TXE	UCR.08	R	**Transmitter empty.** UTX registr je prázdný, vysílání dokončeno.
RXF	UCR.09	RW	**Receiver full.** Příchozí data jsou připravena v registru URX, příjem dokončen. Vlajku je nutno resetovat zápisem do UCR.
LOP	UCR.10	W	**Loopback.** Pokud LOP = 1, je přijímací a vysílací modul zapojen do smyčky, tzn. výstup vysílacího modilu Tx je připojen na vstup Rx přijímacího modulu. Slouží k ověření funkčnosti elektroniky UARTu. Výchozí stav je LOP = 0. Výstupní Tx port mikropočítače není výběrem činnosti ovlivněn.
RSVD	UCR.11	NI[17]	**Reserved.** Čteno jako nula.

Tab. 26: Kontrolní registr

BD1	BD0	$\text{baud} \cdot \text{s}^{-1}$
SYMBOLOVÁ RYCHLOST		
0	0	9600
0	1	19200
1	0	38400
1	1	115200

PR1	PR0	Parita
PARITA		
0	0	Žádná
0	1	Lichá
1	0	Sudá
1	1	Rezerva

Tab .27: Platné hodnoty symbolové rychlosti a parity

[17] NI – Není Implementováno (*Not Implemented*).

URX : UART RECEIVER REGISTER **Zápis: Žádný** **Čtení: 0x7FE**

MSB											LSB
RSVD	RSVD	RSVD	RSVD	D	D	D	D	D	D	D	D

D	UCR.00-07	R	**Data bit.** Datové bity jsou přijímany v pořadí od LSB do MSB.
RSVD	UCR.08-11	NI	**Reserved.** Čteno jako nula.

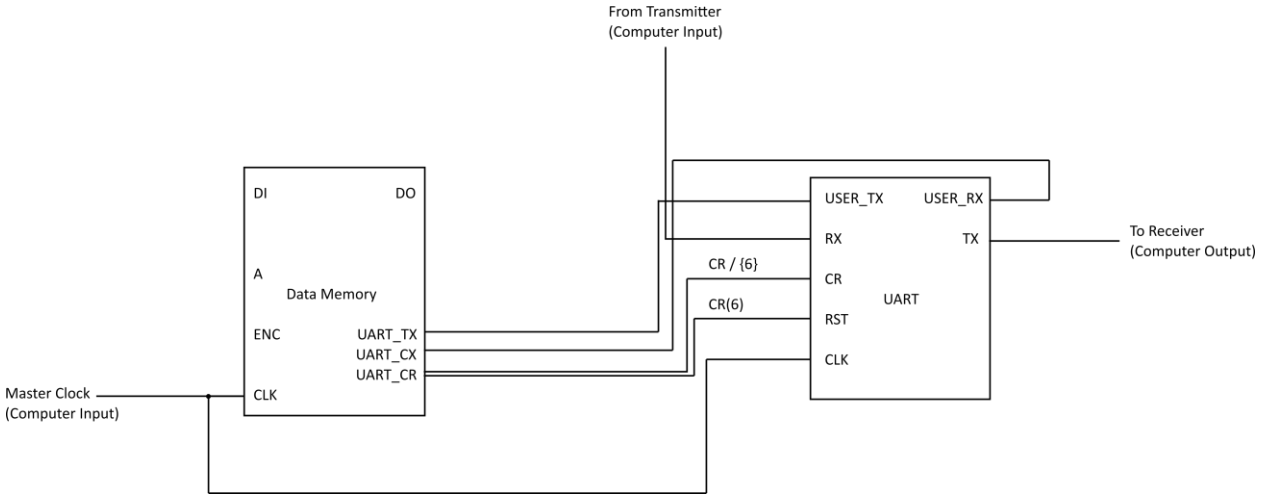
Tab. 28: Přijímací registr

UTX : UART TRANSMITTER REGISTER **Zápis: 0x7FD** **Čtení: Žádný**

MSB											LSB
RSVD	RSVD	RSVD	RSVD	D	D	D	D	D	D	D	D

D	UCR.00-07	W	**Data bit.** Datové bity jsou vysílány v pořadí od LSB do MSB.
RSVD	UCR.08-11	NI	**Reserved.** Čteno jako nula.

Tab. 29: Vysílací registr



Obr. 9: Paměťově mapovaný UART modul

3.2 Časovač

Mikropočítač obsahuje jednoduchý časovač k odměřování zvolených časových úseků. Podobně jako UART je i tento časovač paměťově mapovaný. Zařízení sestává z 32-bitové čítače a komparátoru. Čítač je řízen hlavním hodinovým signálem CLK (Clock) a je tedy inkrementován s každou náběžnou hranou. Takto jsou načítány jednotlivé hodinové pulzy a jejich součet je v každém cyklu porovnán s hodnotou CMPx (Compare) registru. V případě rovnosti je čítač resetován a stavová vlajka DON (Done) je nastavena ve STS (Timer Status) registru. Přesný funkční popis obvodu je patrný z výňatku níže.

```
PROCESS(CLK)
BEGIN
    IF(rising_edge(clk)) THEN
        IF(compare > 0) THEN
            counter <= counter + x"00000001";
            sts(1) <= '0';
            IF(counter = compare) THEN
                sts <= "001";
                counter <= x"00000000";
            END IF;
        ELSE
            sts <= x"002";
            counter <= x" 00000000";
        END IF;
    END IF;
END PROCESS;
```

Tab. 30: Výňatek z mikropočítačového časovač

Čítač je aktivován zapsáním nenulové hodnoty do CMPx registru. Datová šířka mikropočítače je omezena 12-ti bity, CMPx registr je tak nutno rozdělit na tři slova po třech registrech a to, CMPL (Compare Low), CMPM (Compare Medium) a CMPL (Compare Low). Jelikož je k naplnění celého CMPx potřeba třech paměťových zápisů a čítač začíná počítat od první nenulové hodnoty, musí být dodrženo určité pořadí v jakém jsou registry plněny. Jednoduše platí, že zápis do nejvyššího registru CMPH indikuje dokončení komfigurace celého CMPx a čítač může být zapnut či vypnut v závislosti na hodnotě v něm obsažené.

CMPL : COMPARE LOW REGISTER **Zápis: 0x7FB** **Čtení: Žádný**

MSB											LSB
D	D	D	D	D	D	D	D	D	D	D	D

D	CMP.00-07	W	**Data bit.** Dolních 12 *compare* bitů.
RSVD	CMP.08-11	NI	**Reserved.**

Tab. 31: CMPL registr

CMPM : COMPARE MEDIUM REGISTER **Zápis: 0x7FA** **Čtení: Žádný**

MSB											LSB
D	D	D	D	D	D	D	D	D	D	D	D

D	CMP.00-07	W	**Data bit.** Středních 12 *compare* bitů.
RSVD	CMP.08-11	NI	**Reserved.** Čteno jako nula.

Tab. 32: CMPM registr

CMPH : COMPARE HIGH REGISTER **Zápis: 0x7F9** **Čtení: Žádný**

MSB											LSB
D	D	D	D	D	D	D	D	D	D	D	D

D	CMP.00-07	W	**Data bit.** Horních 12 *compare* bitů. Validace CMPx.
RSVD	CMP.08-11	NI	**Reserved.** Čteno jako nula.

Tab. 33: CMPH registr

STS : STATUS REGISTER **Zápis: Žádný** **Čtení: 0x7FC**

MSB											LSB
RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	OVF	EMP	DON

DON	CMP.00	R	**Done.** Požadovaná byla hodnota odměřena. Vlajka je resetována vynulováním CMPx registru.
EMP	CMP.01	R	**Empty.** CMPx registr je prázdný a čítač je zastaven (vypnuto).
OVF	CMP.02	R	**Overflow.** Signalizuje přetečení čítače. Příčinou bývá nastavení větší hodnoty CMPx (36 bitů) než je velikost čítače (32 bitů).
RSVD	CMP.03-11	NI	**Reserved.** Čteno jako nula.

Tab. 34: STS registr

Časovač je velmi důležitou součastí každého výpočetního systému. Integrovaný časovač umožňuje například pomocí přerušení řídit přepínání programového kontextu daného zařízení (*multitasking*).

Kapitola 4

Instrukční sada

4.1 Klasifikace instrukčních sad

Instrukční sada je soubor elementárních a často atomických příkazů, které může daný procesor vykonávat. Rovněž sem patří popis veškerých registrů, paměťí, přerušení a další komponent, na kterých daná sada instrukcí operuje. Instrukční sada neříká nic o interní architektuře procesoru a ani nepopisuje jeho vnitřní stavbu. Rozdílné rodiny procesorů mohou mít velice podobnou či přímo identickou sadu instrukcí, i když se interně velice liší, ať už např. počtem registrů, jejich užitím, adresováním či přímo použitou výrobní technologií. Formováním těchto elementárních strojových příkazů pak vznikají složitější operace. Každá z instrukcí programu je vykonána sekvenčně.
Nezávisle na tom, jak složitá či obsáhlá daná instrukční sada je, podle konvencí by obecně měla vždy obsahovat následující typy instrukcí:

Aritmeticko-logické operace

- Instrukční sada by měla obsahovat instrukce pro součet, případně rozdíl, násobení a dělení. K provedení rozdílu není nezbytně potřeba vlastní instrukce, neboť odečítanou hodnotu lze negovat bit po bitu, výsledek následně inkrementovat pokud $Carry = 1$, a provést součet. Podobně je tomu u multiplikativních a dělících instrukcí, které lze realizovat formou procedur a bitových operací. Tyto komplexní instrukce nicméně velmi usnadňují samotné programování a za určitých okolností mohou být provedeny i rychleji.
- Implementace bitových operací jako logický součet a součin, negace případně exkluzivní součet, operace bitového posunu apod.
- Jednotka většinou podává informace o stavu proběhlé operace, například o přestupu o řád (*Carry* u sčítání či *Borrow* u odečítání), přetečení (*overflow*), nulovém výsledku (*zero*) a jiné. Tyto stavy pak lze

použít u podmíněných instrukcí (*Branch if zero, Branch if carry* apod.). U jednodušších implementací lze pozorovat např. pouze kontrolu nulového akumulátoru a přeskočení instrukce (*Skip if zero)*.

Datové operace

- Instrukce k zápisu konstantních hodnot do registrů a dále instrukce pro výměnu dat mezi registry musí být v instrukční sadě zastoupeny. Pokud dané zařízení obsahuje paměť, je nutná i existence instrukcí pro adresaci a čtení/zápis z a do paměti. V tomto bodě lze pozorovat mezi jednotlivými procesory asi nejmarkatnější rozdíly. Systém může implementovat např. aritmeticko-logické instrukce, které dovolují pracovat s operandy přímo v paměti či výsledky výpočtů do paměti ukládat, což je v kontrastu s *load/store* architekturami, které dovolují operace pouze na registrech a na čtení a zápis z a do paměti jsou vyhrazeny samostatné instrukce.
- Komunikace s venkovními periférierni, tzn. typicky čtení a zápis hodnot z IO portů a sběrnic.

Větvení programu

- Instrukční sada musí též obsahovat instrukce pro nepodmíněné a podmíněné větvení programu. Zde lze rovněž nalézt veliké rozdíly co se komplexnosti týče. Nejprimitivnější podmíněné větvení může zahrnovat např. pouze přeskočení instrukce, pokud je akumulátor nulový, či na základě jiné události signalizující nulový výsledek početní (nebo i vyjímečně datové) operace. Existují však mnohem komplexnější instrukce, které dovolují i dekrementaci registru či paměťové hodnoty, porovnání této dekrementované hodnoty s konstantou, registrem či jinou paměťovou hodnotou a následné větvení na základě nulovosti či nenulovosti výsledku. Taková instrukce tak umožňuje velmi jednoduché tvoření tradičních *for* cyklů známých z vysokoúrovňových programovacích jazyků.
- Na většině architektur lze nalézt instrukce volání procedur, jejichž komplexnost se opět může velmi lišit podle složitosti celkového výpočetního systému. Instrukce v principu sice pouze uloží návratovou adresu do úložiště, kterými může být v nejjednodušším případě registr. Takovýto postup by však umožnil pouze jednoúrovňové volání, což se jeví jako nepraktické. Z tohoto důvodu se používá struktura zásobníku (*stack*), který může být přímo hardwarový, jednoúčelový a jiným instrukcím tudíž nepřístupný, anebo klasický paměťový zásobník. Hardwarový zásobník bývá omezen do řádu několika úrovní (typicky 8, 16 apod.), zatímco paměťový zásobník je omezek pouze velikostí dané paměti či adresováním.

Kategorizace instrukční sady se může ukázat jako velice vágní, nejasná a v realných aplikacích tak lze narazit na spoustu vzájemných kontradikcí, které jsou zapříčiněny nejasnými definicemi. Dá se říci, že mikroprocesorová technika prošla ve vývoji dvěma stádii, a to CISC a RISC přístupy.

CISC (Complex Instruction Set Computer) instrukce

První implementace vycházely z požadavků na malé kapacity a odezvy pamětí. Z těchto důvodů byla požadována veliká hustota kódu a vysoká míra kódování instrukce. CISC instrukční sady proto obsahují veliké množství různých a často i velice specifických instrukcí. Stroj s mnoha komplexními, snadno použitelnými instrukcemi lze velmi jednoduše programovat a míra specifických instrukcí dále umožňuje snadnější návrhy vysokoúrovňových kompilerů. Tyto sady bývají často ortogonální, tzn. že k vykonání

dané úlohy existuje právě jedna instrukce a navíc každá instrukce v sadě může adresovat kterýkoliv z operandů. Ortogonální instrukce nekladou, až na výjimky, žádné nároky na operadny, implicitní operandy tedy neexistují, což je přesně opačný přístup než u RISC či obecně jiných *load*/*store* architektur.
Aritmeticko-logické a jiné operace tak mohou pohodlně kombinovat zdrojové a cílové operandy v registrech, pamětích a konstantách. Jediná instrukce je schopna provést několik delších operací jako je kontrola podmínky, zda instrukce bude provedena, přečtení operandu z paměti, provedení dané akce a zpětný zápis do paměti. Existují tak např. matematické instrukce schopné operovat na maticích, vektorech, nakládat s transcendentními čísly či provádět trigonometrické výpočty. Daní za uživatelskou jednoduchost a kompaktnost instrukcí jsou vysoké nároky na dekódovací logiku procesoru. K zachování vysoké hustoty kódu bývá délka instrukce proměnná a to v závislosti na operandech, což má za následek značně nejednotný formát instrukcí.

RISC (Reduced Instruction Set Computer)

RISC je příkladem zcela protichůdné filozofie. V mnoha aplikacích bylo zjištěno, že takové množství kompexních instrukcí zkrátka není potřeba, případně že je ekonomičtější tuto instrukci „opsat“ sekvencí primitivnějších instrukcí. Hezkou ilustrací takového případu (pokud bychom měli uvést alespoň jednu) je instrukce *CCS – Count, Compare and Skip* použitou v jednom z prvních univerzálních digitálních počítačů,a to AGC[18]:

Code 01. I: CCS K Count, Compare and Skip
QC0 Set c(A) = DABS [b(k)];
Set c(K) = b(K), editing if K is 0020 − 0023.
Take next instruction from I + 1 if b(K) > +0;
from I + 2 if b(K) = +0;
from I + 3 if b(K) < −0;
from I + 4 if b(K) = −0.
Remarks: The Diminished Absolute Value of an integer x
is:

$$\mathrm{DABS}(x) = \begin{cases} |x| - 1 & if\ |x| > 1 \\ +0 & if\ |x| \leq 1 \end{cases}$$

Tento výňatek je doslovný přepis z osmé strany materiálu [9]. CSS tedy přečetla hodnotu z paměti K, spočítala z ní DABS a tuto hodnotu uložila do akumulátoru. V závislosti na tom, zda byla původní hodnota v paměti kladná, záporná, nulová zleva nebo zprava, byla přeskočena jedna, dvě, tři nebo čtyři následující instrukce. Primárním účelem instrukce bylo vytvoření klasického *for* cyklu (pro kladné hodnoty iterační proměnné). Zbytečná komplexnost je patrná v případě, že porovnávaná hodnota nebyla nikdy kladná. Jelikož má jedno instrukční slovo 16 bitů, zůstaly nevyužity celé 4 byty, které se podařilo v optimálním případě vyplnit např. užitečnými konstantami. Pokud by instrukční sada

[18] AGC – Apollo Guidance Computer, 16-bitový navigační počítač primárně pro program Apollo, používán od roku 1966 do roku 1975.

neobsahovala primitivnější kondicionální větvící instrukci (která např. přeskočí pouze jednu jedinou instrukci), hrozilo by, že veliká část paměti zůstane nevyužita, protože CCS počítá s mezerou na 4 instrukční slova.
Podobně nadbytečné se může zdát i ortogonální adresování, i když je pravda, že zrovna relativní nezávislost cílových a zdrojových operandů opět velice usnadňuje programování.

Typická RISC instrukce bude mít tedy jednotný formát ve všech případech; každá bitová pozice tak nese vždy, nebo až na drobné výjimky, stejnou informaci nezávisle na typu instrukce. Takovýto přístup vede ke značnému zjednodušení dekódovací logiky a celkové přehlednosti architektury. RISC procesor bude rovněž obsahovat větší množství identických registrů, ať už těch „skutečných" nebo paměťově adresovatelných. Nejtypičtější charakteristikou však bude jednoduchost a přímočarost každé z instrukcí a hlavně striktní oddělení paměťově orientovaných instrukcí od ostatních (*load-store* architektura), kdy aritmetické a jiné operace jsou prováděny na registrech a výsledky jsou zapisovány až dedikovanými instrukcemi.

4.2 Instrukční slovo

Instrukční slovo je fundamentální *jednotkou* programu. Každá instrukce je kódována právě 16 bity (bity 15 – 0 programového slova) a z perspektivy uživatele se chová atomicky. To znamené, že instrukce na adrese $[\mathrm{PC}+1]$ nebude přečtena ani vykonána, dokud nebude dokončena instrukce $[\mathrm{PC}]$ a to i v případě, že je vlastní instrukční fáze několik hodinových cyklů dlouhá. Historicky některé výpočetní systémy umožňovali u zvláště časově náročných instrukcí (typicky dělení) pokračovat v běhu programu, zatímco byla daná operace prováděna. S takovýmto paralelismem je možné se setkat i v současnosti, zmiňme například známou koexistenci matematického koprocesoru a centrální procesorové jednotky architektury x86.
Přesné zakódování instrukce závisí na jejím typu, nicméně u všech instrukcí je dodrženo jednotné schéma typu *konstanta-operand-opkód*[19].

MSB

IMM/MOD	IMM	IMM	IMM	IMM	IMM	IMM	IMM
IMM	IMM/ZE	IMM/YE	IMM/EXT	OPR/CC	OP2	OP1	OP0

LSB

OP0	IR.00	Bit operačního kódu instrukce.
OP1	IR.01	Bit operačního kódu instrukce.
OP2	IR.02	Bit operačního kódu instrukce.
OPR/CC	IR.03	Operand nebo condicionální kód (*Condition Code*) instrukce.
IMM	IR.04	Extend nebo *immediate* bit. Extend značí rozšířenou instrukční sadu.
IMM	IR.05	Carry Enable nebo *immediate* bit. YE je kondicionální kód.
IMM	IR.06	Zero Enable nebo *immediate* bit. ZE je kondicionální kód.
IMM	IR.07	*Immediate* bit.

[19] Operační kód, neboli *opkód*, specifikuje typ dané instrukce, například součet ADD či datový přesun MOV.

IMM	IR.08	*Immediate* bit.
IMM	IR.09	*Immediate* bit.
IMM	IR.10	*Immediate* bit.
IMM	IR.11	*Immediate* bit.
IMM	IR.12	*Immediate* bit.
IMM	IR.13	*Immediate* bit.
IMM	IR.14	*Immediate* bit.
IMM/MOD	IR.15	*Immediate* nebo MOD bit. MOD (*Mode*) upřesňuje typ Extend instrukce (SLL x SLR).

Tab. 35: Formát instrukčního slova

Existují čtyři typy instrukcí, které přímo ovlivňují přesný výklad jednotlivých polí instukčního slova, jak je popsáno v tabulce níže. Instrukce lze obecně rozdělit na základě komponenty, která ji vykonává, a to aritmeticko logické (ALU), datové (DTU) a větvící či jiné (interní).

Datové pole:	OR.3 Význam	OR.4-6 Význam
ALU Instrukce:	OPR = 1 Zdrojový operand IMM	Okamžitá (*Immediate*) hodnota
	OPR = 0 Zdrojový operand B	Původní význam polí
PIO Instrukce:	OPR = 1 Cílový operand A	Původní význam polí
	OPR = 0 Cílový operand IO	Původní význam polí
MOV IMM Instrukce:	OPR = 0 Cílový operand A	Okamžitá (*Immediate*) hodnota
	OPR = 1 Cílový operand B	Okamžitá (*Immediate*) hodnota
MOV R Instrukce:	OPR = 0 Zdrojový operand A	Původní význam polí
	OPR = 1 Zdrojový operand B	Původní význam polí
BCC Instrukce:	CC = 0 Nepodmíněné vetvení	Okamžitá (*Immediate*) adresa
	CC = 1 Větvení pokud A = 0	Okamžitá (*Immediate*) adresa

OR.OPR/CC Význam		OR.EXT Význam	OR.ZE Význam	OR.YE Význam
0	Pouze Aux B	Extend	Zero CC Enable	Carry CC Enable
1	*Immediate*	*Immediate*/Ignor.	*Immediate*/Ignor.	*Immediate*/Ignor.

Tab. 36: Interpretace instrukčního slova v závislosti na typu instrukce

Pokud je jeden z registrů zvolen jako cílový operand, druhý je automaticky zvolen jako zdrojový, tzn. pokud A je zdrojový operand tak potom B je cílový operand. Vskutku – registry jsou pouze dva, k zakódování operandu tak stačí jediný bit. Datové instrukce se shodnými operandy mají význam prázdné operace (NOP – *No Operation*, viz instrukce B .0 v oddílu 4.5). Z dostupných aritmeticko logických instrukcí by se jako užitečné mohli jevit pouze ADD a MPY. Nicméně první jmenovaný případ je ekvivalentní s levým logickým posunem o 1 bit (SLL #1), jde tedy spíše o *alias*[20] instrukce. V druhém případě by došlo k umocnění hodnoty akumulátoru.

[20] O *aliasu* hovoříme tehdy, máme-li dvě instrukce s rozdílnými mnemonikami nebo kódováním, které však provádějí zcela totožnou operaci. Přirozeně je nesmyslné, aby instrukční sada obsahovala funkčně ekvivaletní instrukce s rozdílným kódováním. Alternativní mnemonické označení některých instrukcí však může zdrojový kód zpřehlednit, příkladem budiž dvojice B .0 a NOP.

4.3 Přehled instrukcí

Instrukční sada mikropočítače čítá 18 instrukcí v několika různých obměnách. Krátký soupis všech dostupných instrukcí je dostupný v tabulce níže, včetně kódování. V následujícím oddílu pak lze nalézt detailní popis každé individuální instrukce.

OP/Mnemonic	Operand	Description	C	Bytes	EXT	CE	FA	Comp
ARITHMETIC AND LOGICAL OPERATIONS								
000/ADD	Aux B	Add registers	1	2	No	Yes	Yes	ALU
000/ADD	IMM	Add immediate	1	2	No	No	Yes	ALU
001/NOT	Aux B (implicit)	Negate Aux B	1	2	No	Yes	Yes	ALU
010/AND	Aux B	Bitwise AND registers	1	2	No	Yes	Yes	ALU
010/AND	IMM	Bitwise AND immediate	1	2	No	No	Yes	ALU
011/OR	Aux B	Bitwise OR registers	1	2	No	Yes	Yes	ALU
011/OR	IMM	Bitwise OR immediate	1	2	No	No	Yes	ALU
DATA TRANSFER OPERATIONS								
100/MOV A	IMM	Move immediate	1	2	No	No	No	DTU
100/MOV B	IMM	Move immediate	1	2	No	No	No	DTU
101/MOV A	Aux B (implicit)	Move registers	1	2	No	Yes	No	DTU
101/MOV B	A (implicit)	Move registers	1	2	No	Yes	No	DTU
001/LOD	IMM	Load Accumulator	1	2	No	No	No	DTU
001/STO	IMM	Store Accummulator	1	2	No	No	No	DTU
INPUT/OUTPUT OPERATIONS								
110/PIO	IO	Process Input and Output	1	2	No	Yes	No	DTU
110/PIO	A	Process Input and Output	1	2	No	Yes	No	DTU
TRANSFER CONTROL OPERATIONS								
111/B	IMM	Branch to immediate	1	2	No	No	No	Interní
111/BEQ	IMM	Branch to immediate if A = 0	1	2	No	Yes	No	Interní

OP/Mnemonic	Operand	Description	C	Bytes	EXT	CE	FA	Comp
ARITHMETIC AND LOGICAL OPERATIONS – EXTEND								
000/SLL	IMM	Shift logical left	1	2	Yes	Yes	Yes	ALU
000/SLR	IMM	Shift logical right	1	2	Yes	Yes	Yes	ALU
010/MPY	AB (implicit)	Multiply registers	2	2	Yes	Yes	Yes	ALU
010/DIV	AB (implicit)	Divide registers	1	2	Yes	Yes	Yes	ALU
DATA TRANSFER OPERATIONS – EXTEND								
101/STO	Aux B	Store A to pointer [B]	1	2	Yes	Yes	No	DTU
101/LOD	Aux B	Load A from pointer [B]	1	2	Yes	Yes	No	DTU
INPUT/OUTPUT OPERATIONS – EXTEND								
001/CHHP	IMM	Change high half port	1	2	Yes	No	No	DTU
001/CLHP	IMM	Change low half port	1	2	Yes	No	No	DTU
TRANSFER CONTROL OPERATIONS – EXTEND								
011/CALL	IMM	Call procedure by address	1	2	Yes	No	No	Interní
011/RET	None	Return from procedure	1	2	Yes	Yes	No	Interní

OP/Mnemonic	Operand	Description	C	Bytes	EXT	CE	FA	Comp
TRANSFER CONTROL OPERATIONS – EXTEND (cont.)								
110/DAB	IMM (IMM)	Decrement A and branch	4	2	Yes	No	Yes	Sekv[21]
000/CMS	IMM	Change memory segment	1	2	Yes	No	No	Interní

Tab. 37: Přehled instrukcí mikropočítače[22]

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

Tab. 38: Přehled kódování jednotlivých instrukcí mikropočítače

[21] Sekvence interních, datových a aritmeticko logických instrukcí.

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

4.4 Definice instrukcí

ADD B

Funkce:	Součet
Popis:	ADD B přičte hodnotu v pomocném registru B k hodnotě akumulátoru a výsledek zapíše do akumulátoru. Všechny numerické hodnoty jsou zadávány jako kladné, bez znaménka (*unsigned*).
Příklad:	Akumulátor obsahuje hodnotu 0x01 a registr B obsahuje hodnotu 0x02. Instrukce, ADD B oba registry sečte a výsledek součtu zapíše do akumulátoru, který tak bude obsahovat hodnotu 0x03.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 0 0 0 0
Operace:	$A \leftarrow A + B$ $CS \leftarrow 0, FS \leftarrow 0, OS \leftarrow 1$

ADD IMM

Funkce:	Součet
Popis:	ADD IMM připočte okamžitou hodnotu k akumulátoru a výsledek zapíše do akumulátoru. Všechny numerické hodnoty jsou zadávány jako kladné, bez znaménka (*unsigned*).
Příklad:	Akumulátor obsahuje hodnotu 0x01 a kódovaná okamžitá hodnota je 0x02. Instrukce, ADD #2 provede součet a výsledek zapsaný v akumulátoru tak bude 0x03.
Bytů:	2
Cyklů:	1
Kódování:	D D D D | D D D D | D D D D | 1 0 0 0
Operace:	$A \leftarrow A + \text{Immediate data}$ $CS \leftarrow 0, FS \leftarrow 0, OS \leftarrow 3$

NOT B

Funkce:	Negace
Popis:	NOT invertuje každý z bitů pomocného registru B a výsledek zapíše zpět do pomocného registru. Instrukce vynuluje CF.
Příklad:	Pomocný registr B obsahuje hodnotu 0x0A (1010b). Instrukce, NOT B provede negaci a do registru B tak bude zapsána hodnota 0x05 (0101b). Operand B je implicitní.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 1 0 0 1
Operace:	$B \leftarrow \text{NOT } B$ $CS \leftarrow 0, FS \leftarrow 1, OS \leftarrow 1$

LOD IMM

Funkce:	Datový přesun
Popis:	LOD zkopíruje hodnotu z datové paměti na adrese $[A]$ do akumulátoru. Zdrojová data nejsou touto operací ovlivněna.
Příklad:	Datová paměť obsahuje na adrese 0x05 hodnotu 0x0A. Instrukce, LOD #0x005 zapíše 0x00A do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	0 A A A | A A A A | A A A A | 0 0 0 1
Operace:	$A \leftarrow C[A]$ $CS \leftarrow 1, FS \leftarrow 0, OS \leftarrow 2, AS \leftarrow 0$

STO IMM

Funkce:	Datový přesun
Popis:	STO A zkopíruje hodnotu z akumulátoru na adresu $[A]$ do datové paměti. Zdrojový operand není touto operací ovlivněn.
Příklad:	Akumulátor obsahuje hodnotu 0x00A. Instrukce, STO #0x005 zapíše 0x0A na adresu 0x05 do datové paměti.
Bytů:	2
Cyklů:	1
Kódování:	1 A A A | A A A A | A A A A | 0 0 0 1
Operace:	$C[A] \leftarrow A$ $CS \leftarrow 1, FS \leftarrow 0, OS \leftarrow 0, ENA \leftarrow 1, ENB \leftarrow 1$[23]$, AS \leftarrow 0$

[23] Instrukce STO jako jediná může použít signály ENA/ENB z Write Back fáze jako signály dekódovací sekce. Viz oddíl 2.5.

AND B

Funkce:	Bitový AND
Popis:	AND provede bitový součin hodnot akumulátoru a pomocného registru B. Výsledek je zapsán zpět do akumulátoru.
Příklad:	Akumulátor obsahuje hodnotu 0x00F (1111b) a B registr hodnotu 0x00A (1010b). Instrukce, AND B zapíše 0x0A (1010b) do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 0 0 1 0
Operace:	$A \leftarrow A \wedge B$ $CS \leftarrow 0, FS \leftarrow 2, OS \leftarrow 1$

AND IMM

Funkce:	Bitový AND
Popis:	AND provede bitový součin hodnot akumulátoru a kódované konstanty. Výsledek je zapsán zpět do akumulátoru.
Příklad:	Akumulátor obsahuje hodnotu 0x0F (1111b) a kódovaná okamžitá hodnota je 0x0A (1010b). Instrukce AND #0x0A zapíše 0x0A (1010b) do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	D D D D | D D D D | D D D D | 1 0 1 0
Operace:	$A \leftarrow A \wedge \text{Immediate data}$ $CS \leftarrow 0, FS \leftarrow 2, OS \leftarrow 3$

OR B

Funkce:	Bitový OR
Popis:	OR provede bitový součin hodnot akumulátoru a pomocného registru B. Výsledek je zapsán zpět do akumulátoru.
Příklad:	Akumulátor obsahuje hodnotu 0x0C (1100b) a registr B obsahuje hodnotu 0x03 (0011b). Instrukce, OR B zapíše 0x0F (1111b) do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 0 0 1 1
Operace:	$A \leftarrow A \vee B$ $CS \leftarrow 0, FS \leftarrow 3, OS \leftarrow 1$

OR IMM

Funkce: Bitový OR

Popis: OR provede bitový součet hodnot akumulátoru a kódované konstanty. Výsledek je zapsán zpět do akumulátoru.

Příklad: Akumulátor obsahuje hodnotu 0x0C (1100b) a kódovaná okamžitá hodnota je 0x03 (0011b). Instrukce,

OR #3

zapíše 0x0F (1111b) do akumulátoru.

Bytů: 2

Cyklů: 1

Kódování:

D	D	D	D	D	D	D	D	D	D	D	D	1	0	1	1

Operace: $A \leftarrow A \vee \text{Immediate data}$
$CS \leftarrow 0, FS \leftarrow 3, OS \leftarrow 3$

MOV A, IMM

Funkce: Datový přesun

Popis: MOV zapíše okamžitou hodnotu do akumulátoru.

Příklad: Kódovaná konstanta je 0x0A. Instrukce,

MOV A, #0x0A

zapíše do akumulátoru hodnotu 0x0A.

Bytů: 2

Cyklů: 1

Kódování:

D	D	D	D	D	D	D	D	D	D	D	D	0	1	0	0

Operace: $A \leftarrow \text{Immediate data}$
$CS \leftarrow 1, FS \leftarrow 0, OS \leftarrow 3$

MOV B, IMM

Funkce: Datový přesun

Popis: MOV zapíše okamžitou hodnotu do pomocného registru B.

Příklad: Kódovaná konstanta je 0x0A. Instrukce,

MOV B, #0x0A

zapíše do registru B hodnotu 0x0A.

Bytů: 2

Cyklů: 1

Kódování:

D	D	D	D	D	D	D	D	D	D	D	D	1	1	0	0

Operace: $B \leftarrow \text{Immediate data}$
$CS \leftarrow 1, FS \leftarrow 0, OS \leftarrow 3$

MOV A, B

Funkce:	Datový přesun
Popis:	MOV zkopíruje obsah pomocného registru B do akumulátoru. Původní hodnota zdrojového operandu zůstává zachována.
Příklad:	Registr B obsahuje hodnotu 0x0A. Instrukce $\mathrm{MOV\ A, B}$ zapíše 0x0A do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 0 1 0 1
Operace:	$\mathrm{A} \leftarrow \mathrm{B}$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 0, \mathrm{OS} \leftarrow 0$

MOV B, A

Funkce:	Datový přesun
Popis:	MOV zkopíruje obsah akumulátoru do pomocného registru B. Původní hodnota zdrojového operandu zůstává zachována.
Příklad:	Akumulátor obsahuje hodnotu 0x0A. Instrukce, $\mathrm{MOV\ B, A}$ zapíše 0x0A do registru B.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 1 1 0 1
Operace:	$\mathrm{B} \leftarrow \mathrm{A}$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 0, \mathrm{OS} \leftarrow 1$

PIO IO

Funkce:	Komunikace s externími periférierni
Popis:	PIO IO zapíše obsah akumulátoru na výstupní piny IO portu. Směr komunikace je volen pomocí instrukcí CHHP a CLHP, viz dále. IO port se chová jako registr a jeho výstup tak zůstane nezměněn i pokud je již vykonávána jiná instrukce nebo hodnota Akumulátoru byla změna. Hodnotu na IO portu tak lze změnit pouze další PIO instrukcí.
Příklad:	Akumulátor obsauje hodnotu 0x0BA, bity $\mathrm{IO}[3-0]$ jsou nakonfigurovány jako výstup a bity $\mathrm{IO}[11-4]$ jako vstup. Instrukce, $\mathrm{PIO\ IO}$ zapíše 0x00A do IO portu a tato hodota se tak objeví na IO pinech a zůstane tam do této doby, dokud nebude další PIO instrukcí přepsána. Horních 8 bitů akumulátoru je ignorováno.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 0 1 1 0
Operace:	$\mathrm{IO} \leftarrow \mathrm{A}$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 1, \mathrm{OS} \leftarrow 0, \mathrm{END} \leftarrow 1$

PIO A

Funkce:	Komunikace s externími periférierni
Popis:	PIO A zkopíruje hodnotu ze vstupních IO pinů do akumulátoru. Směr komunikaci je volen pomocí instrukcí CHHP a CLHP, viz dále. Po celou dobu trvání instrukce se hodnota IO nesmí měnit, jinak dojde k zapsání neplatných dat.
Příklad:	Hodnota IO portu je 0x0BA, bity $\mathrm{IO}[3-0]$ jsou nakonfigurovány jako výstup a bity $\mathrm{IO}[11-4]$ jako vstup. Instrukce, PIO A zapíše0x00B do Akumulátoru. Spodní 4 bity IO portu jsou ignorovány.
Bytů:	2
Cyklů:	1
Kódování:	X X X X | X X X X | X Z Y 0 | 1 1 1 0
Operace:	$\mathrm{A} \leftarrow \mathrm{IO}$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 2, \mathrm{OS} \leftarrow 1, \mathrm{END} \leftarrow 1$

B

Funkce:	Nepodmíněné větvení
Popis:	B provede nepodmíněný skok na adresu v programové paměti. Tato adresa je konstatní a je relativní vůči programovému čítači. Nejvyšší bit adresy, tj. A[11] udává směr větvení, tzn. zda k bude adresa k PC přičtena nebo naopak odečtena. Tímto způsobem je možno přeskočit 2047 instrukcí nebo se o 2047 instrukcí vrátit v programu nazpět. Odečet adresy je proveden invertováním bitů adresy A[10-0], přičtením k PC a jeho následnou inkrementací, pokud došlo k přestupu řádu[24] (*carry*). Instrukce B .0 může být rovněž použita jako NOP pro účely např. časování, neboť nemá žádný vliv na činnost procesoru (k PC bude přičtena 0).
Příklad:	PC obsahuje hodnotu 0x0020 a adresa v paměti nese název .label. Instrukce, B .label .label: přičte k PC kladnou hodnotu adresy a v dalším instrukčním cyklu tak bude přečtena a vykonána instrukce právě na této adrese. Pokud je tedy .label vzdálen například 5 bytů od současné instrukce, bude do PC uložena hodnota 0x0025. Alternativně instrukce, B .-1 sníží PC o jedničku a mikropočítač tak přejde do nekonečné smyčky.
Bytů:	2
Cyklů:	1
Kódování:	A A A A | A A A A | A A A A | 0 1 1 1
Operace:	$\text{if } (\mathrm{A}[11] = 0) \text{ then } \mathrm{PC} \leftarrow \mathrm{PC} + \text{Immediate address}$ $\text{else } \mathrm{PC} \leftarrow \mathrm{PC} + \text{NOT Immediate address}$ $\text{if}(\text{Carry} = 1)\text{then } \mathrm{PC} \leftarrow \mathrm{PC} + 1$

[24] One's complement.

BEQ

Funkce: Podmíněné větvení

Popis: BEQ provede podmíněný skok na adresu v programové paměti. Chování této instrukce je totožné s B instrukcí s tím rozdílem, že BEQ bude vykonán pouze pokud je hodnota v akumulátoru nulová[25]. V opačném případě je tato instrukce ignorována.

Příklad: PC obsahuje hodnotu 0x0020, Akumulátor je vynulován a .label je adresa v programové paměti. Instrukce,

```
BEQ .label
.label:
```

provede skok na adresu .label podle stejného schématu jako výše, protože podmínka nulového akumulátoru je splněna. Pokud by akumulátor obsahoval nenulovou hodnotu, programový čítač PC by byl pouze ve S_1 Fetch stavu inkrementován, došlo by tak k vykonání instrukce na adrese 0x0021.

Bytů: 2

Cyklů: 1

Kódování:

A	A	A	A	A	A	A	A	A	A	A	A	1	1	1	1

Operace:

$$\text{if}(\text{ZF} = 1)\ \text{then}$$
$$\quad \text{if}\ (\text{A}[3] = 0)\ \text{then PC} \leftarrow \text{PC} + \text{Immediate address}$$
$$\quad \text{else PC} \leftarrow \text{PC} + \text{NOT Immediate address}$$
$$\quad\quad \text{if}(\text{Carry} = 1)\text{then PC} \leftarrow \text{PC} + 1$$

SLL IMM

Funkce: Levý logický bitový posun

Popis: SLL IMM posune obsah akumulátoru vlevo o daný počet bitů specifikovaný operandem. MSB bit je při posunu zahozen a LSB bit je nahrazen bitem nulovým. SLL je ekvivaletní k operaci násobení celočíselnou mocninou o základu 2, tedy 2^{IMM}. Instrukce je však realizována skutečným posunem, nikoli součinem.

Příklad: Akumulátor obsahuje hodnotu 0x001 a kódovaná okamžitá hodnota je 0x004. Instrukce,

```
SLL #4
```

posune obsah akumulátoru o 4 bitové pozice doleva. Hodnota akumulátoru bude nyní 0x010.

Bytů: 2

Cyklů: 1

Kódování:

0	X	X	X	D	D	D	D	X	Z	Y	1	0	0	0	0

Platí pro $\text{Z} \neq \text{Y}$.

Operace:

$$\text{A} \leftarrow \text{A} \ll \text{Immediate data}$$

Ekvivalentně: $\text{A} \leftarrow \text{A} \cdot 2^{\text{Immediate data}}, \text{A} \in \mathbb{Z}$

$$\text{CS} \leftarrow 0, \text{FS} \leftarrow 4, \text{OS} \leftarrow 3$$

[25] Existuje rozdíl mezi nulovou (*empty*) a vyprázněnou (*truncated*) hodnotou akumulátoru, viz poznámka na konci oddílu.

SLR IMM

Funkce: Pravý logický bitový posun

Popis: SLR IMM posune obsah akumulátoru vpravo o daný počet bitů specifikovaný operandem. LSB bit je při posunu zahozen a MSB bit je nahrazen bitem nulovým. SLR je ekvivaletní k operaci dělení celočíselnou mocninou o základu 2, tedy 2^{IMM}. Instrukce je však realizována skutečným posunem, nikoli podílem.

Příklad: Akumulátor obsahuje hodnotu 0x010 a kódovaná okamžitá hodnota je 0x004. Instrukce,

SLR #4

posune obsah akumulátoru o 4 bitové pozice doprava. Hodnota akumulátoru bude nyní 0x001.

Bytů: 2

Cyklů: 1

Kódování:

1	X	X	X	D	D	D	D	X	Z	Y	1	0	0	0	0

Platí pro $Z \neq Y$.

Operace: $A \leftarrow A \gg \text{Immediate data}$

Ekvivalentně: $A \leftarrow A \frac{1}{2^{\text{Immediate data}}}, A \in \mathbb{Z}$

$CS \leftarrow 0, FS \leftarrow 5, OS \leftarrow 3$

MPY

Funkce: Součin

Popis: MPY provede součin registrů A a B. Výsledek je 24-bitové číslo a proto má instrukce dvě write-back fáze (viz oddíl 2.4). V první fázi je spodních 12 bitů zapsáno do akumulátoru, horních 12 bitů je pak ve druhé fázi zapsáno do pomocného registru. Všechny numerické hodnoty jsou bez znaménka (*unsigned*).

Příklad: Akumulátor obsahuje hodnotu 0x123 a register B hodnotu 0x456. Instrukce,

MPY

provede součin, akumulátor tak bude obsahovat hodnotu 0xDC2 a reigstr B hodnotu 0x04E. Součin obou obou tedy je 0x4EDC2.

Bytů: 2

Cyklů: 2

Kódování:

0	X	X	X	X	X	X	X	X	Z	Y	1	0	0	1	0

Operace: $A \leftarrow A \cdot B$

$CS \leftarrow 0, FS \leftarrow 6, OS \leftarrow 1$

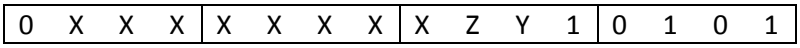
DIV

Funkce:	Podíl
Popis:	DIV podělí akumulátor registrem B a výsledek zapíše zpět do akumulátoru. Pokud jsou hodnoty obsažené v registrech nesoudělné, stavová vlajka REM (CF) je nastavena a v akumulátoru je ponechán částečný výsledek. Dělení je realizováno pomocí „restoračního“ algoritmu (*Restoring Division*) a je dokončeno během jediného cyklu. Tento jednoduchý postup využívá pouze sčítacích (odčítacích) a komparačních obvodů.
Příklad:	Akumulátor obsahuje hodnotu 0x456 a registr B hodnotu 0x123. Instrukce, DIV registry mezi sebou podělí. Jelikož jsou dodané hodnoty nesoudělné, REM (CF) bude nastavena na log. 1 a v akumulátoru bude zanechán částečný podíl, tedy 0x003.
Bytů:	2
Cyklů:	1
Kódování:	1 X X X X X X X X Z Y 1 0 0 1 0
Operace:	$A \leftarrow A/B$ $CS \leftarrow 0, FS \leftarrow 7, OS \leftarrow 1$

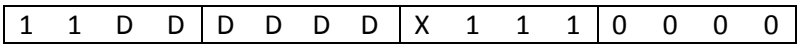
STO B

Funkce:	Datový přesun
Popis:	STO zkopíruje hodnotu z akumulátoru na adresu $[B]$ do datové paměti, kde B je pomocný registr B. Registr B tu zde tudíž plní funkci ukazetele (*pointeru*) a umožňuje tak snadno přesouvat části datové paměti. Zdrojové operandy nejsou touto operací ovlivněny.
Příklad:	Akumulátor obsahuje hodnotu 0x00A a registr B hodnotu 0x005. Instrukce, STO B zapíše 0x00A na adresu 0x005 do datové paměti.
Bytů:	2
Cyklů:	1
Kódování:	1 X X X X X X X X Z Y 1 0 1 0 1
Operace:	$C[B] \leftarrow A$ $CS \leftarrow 1, FS \leftarrow 0, OS \leftarrow 1, ENA \leftarrow 1, ENB \leftarrow 1, AS \leftarrow 1$

LOD B

Funkce:	Datový přesun
Popis:	STO zkopíruje hodnotu z datové paměti na adrese $[\mathrm{B}]$ do akumulátoru, kde B je pomocný registr B. Registr B tu zde tudíž plní funkci ukazetele (*pointeru*) a umožňuje tak snadno přesouvat části datové paměti. Zdrojové operandy nejsou touto operací ovlivněny.
Příklad:	Datová paměť na adrese 0x005 obsahuje hodnotu 0x00A a do registru B je uložena hodnota 0x005. Instrukce, $\mathrm{STO\ B}$ zapíše 0x00A do akumulátoru.
Bytů:	2
Cyklů:	1
Kódování:	
Operace:	$\mathrm{A} \leftarrow \mathrm{C}[\mathrm{B}]$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 0, \mathrm{OS} \leftarrow 2, \mathrm{AS} \leftarrow 1$

CHHP IMM

Funkce:	Konfigurace IO portu
Popis:	CHHP konfiguruje piny $\mathrm{IO}\ [11-6]$ vstupně výstupního portu. Směr komunikace je volen pomocí signálu IOCFG (viz oddíl 2.3.4) a to následovně: • Pokud $\mathrm{IMM}[i] = 0$, bude i-tý IO pin nastaven jako vstupní. • Pokud $\mathrm{IMM}[i] = 1$, bude i-tý IO pin nastaven jako výstupní.
Příklad:	Chceme-li nastavit piny $\mathrm{IO}[3-0]$ jako výstupní a piny $\mathrm{IO}[11-4]$ jako vstupní pak hodnota konfiguračního signálu IOCFG musí být 0xFF0. Instrukce, $\mathrm{CHHP\ \#0x3F}$ nejprve označí horních 6 bitů IO portu jako vstupy. Zbylých spodních 6 bitů je nutno nakonfigurovat pomocí CLHP instrukce, viz níže.
Bytů:	2
Cyklů:	1
Kódování:	 Platí pro $\mathrm{Z} = \mathrm{Y}$.
Operace:	$\mathrm{IOCFG}[11-6] \leftarrow \mathrm{Immediate\ data}[9-4]$ $\mathrm{CS} \leftarrow 1, \mathrm{FS} \leftarrow 3, \mathrm{OS} \leftarrow 3, \mathrm{END} \leftarrow 1$

CHLP IMM

Funkce:	Konfigurace IO portu
Popis:	CHHP konfiguruje piny $\mathrm{IO}\ [5-0]$ vstupně výstupního portu. Směr komunikace je volen pomocí signálu IOCFG (viz oddíl 2.3.4) a to následovně: • Pokud $\mathrm{IMM}[i] = 0$, bude i-tý IO pin nastaven jako vstupní. • Pokud $\mathrm{IMM}[i] = 1$, bude i-tý IO pin nastaven jako výstupní.
Příklad:	Chceme-li nastavit piny $\mathrm{IO}[3-0]$ jako výstupní a piny $\mathrm{IO}[11-4]$ jako vstupní pak hodnota konfiguračního signálu IOCFG musí být 0xFF0. Instrukce, $\mathrm{CHLP\ \#0x30}$

nejprve označí spodních 6 bitů IO portu jako vstupy. Zbylých spodních 6 bitů je nutno nakonfigurovat pomocí CHHP instrukce, viz výše.

Bytů: 2

Cyklů: 1

Kódování:

1	0	D	D	D	D	D	D	X	1	1	1	0	0	0	0

Platí pro $Z = Y$.

Operace: $\text{IOCFG}[5-0] \leftarrow \text{Immediate data}[9-4]$
$\text{CS} \leftarrow 1, \text{FS} \leftarrow 4, \text{OS} \leftarrow 3, \text{END} \leftarrow 1$

CALL IMM

Funkce: Volání procedury

Popis: CALL provede nepodmíněné volání procedury na adrese specifikované operandem. Narozdíl od větvící instrukce Bcc je adresa kódována absolutně. Tato adresa udává 10-bitový offset daného segmentu paměti, samotný segment je nutno předem zvolit pomocí instrukce CSM, viz dále. Cílová absolutní adresa je pak spočtena následovně:

$$\text{Absolutní adresa} = \text{Segment} \cdot \text{Délka segmentu} + \text{Offset}$$

kde $\text{Délka segmentu} = 2048$. Zároveň je do zásobníku (viz oddíl 2.5) uložena současná (ve stavu S_1 inkrementovaná) hodnota programového čítače PC, která již odkazuje na následující instrukci. Výchozí hodnota segmentu paměti po zapnutí mikropočítače je nedefinovaná a první CALL tak musí předcházen CMS instrukcí. Pokud se volané procedury nacházejí ve stejném programového segmentu, není nutné CMS před každým voláním opakovat. Tento přístup je však silně doporučován z důvodu údržby a přehlednosti zdrojových programů. Při změně délky kódu se přirozeně mění (pousouvají) volané adresy a daná procedura (procedury) se tak mohou snadno dostat mimo rozsah předpokládaného segmentu.
Celkem je možné vykonat až 16 vnořených volání podprogramu. Maximální počet volání je stanoven omezenou hloubkou adresového zásobníku.

Příklad: Programový čítač PC obsahuje hodnotu 0x1234 a volána procedura je na adrese 0x5678. Právě vykonávaná instrukce CALL se tudíž musí nacházet na adrese 0x1233 (signál LOAD). Z rovnice pro výpočet[26] absolutní adresy platí, že segment cílové adresy bude

$$\text{Segment} = \frac{\text{Absolutní adresa}}{\text{Délka segmentu}}_{\text{Offset}=0} = \frac{0\text{x}5678}{0\text{x}400} = 0\text{x}15$$

Zpětným dosazením vypočtené hodnoty segmentu do rovnice absolutní adresy

[26] Pro hledaný programový segment je nejprve nutno položit $\text{Offset} = 0$. Podíl rovněž pracuje pouze s celočíselnými hodnotami. Segment je tudíž částečný podíl a offset pouze zbytek podílu. Pokud ve výsledné adrese obdržíme $\text{Offset} = 0$, říkáme že adresa je segmentově „zarovnána“ (*aligned*).

obdržíme konečně i hledaný offset:

$$\text{Offset} = \text{Absolutní adresa} - \text{Segment} \cdot \text{Délka segmentu}$$
$$= 0\text{x}5678 - 0\text{x}15 \cdot 0\text{x}400 = 0\text{x}278$$

Nyní můžeme provést volaní podprogramu jako

```
CMS #0x15
CALL .0x278
```

Zároveň je na adresu [SP] uložena současná hodnota programového čítače PC, a to 0x1234. Zasobníkový ukazatel SP je přitom navýšen o jedničku. Z dodaných hodnot offsetu a segmentu vypočte kontrolní logika výše zmíněným postupem absolutní adresu a ta je zkopírována do PC. Tímto je CALL instrukce ukončena a v příštím instrukčním cyklu bude již vykonána instrukce na volané adrese.

Bytů: 2

Cyklů: 1

Kódování:

0	A	A	A	A	A	A	A	A	A	A	1	0	0	1	1

Operace:

$$\text{Absolute address} \leftarrow \text{Segment} \cdot 2048 + \text{Immediate data}$$
$$[\text{SP}] \leftarrow \text{PC}$$
$$\text{SP} \leftarrow \text{SP} + 1$$
$$\text{PC} \leftarrow \text{Absolute address}$$

RET

Funkce: Návrat z procedury

Popis: RET zkopíruje návratovou adresu z vrchu zásobníku do programového čítače PC. Zásobníkový ukazatel SP je následně snížen o jedničku. Program tak bude pokračovat na adrese bezprostředně po CALL instrukci, která volání původně vyvolala.

Příklad: Programový čítač obsahuje hodnotu 0x5679. Právě vykonánvaná instrukce RET tudíž musí nacházet na adrese 0x5678. Jestliže,

```
1234 4F13 CALL .0x278
1235 8017 BEQ .-1
...
5678 8013 RET
```

kde první čtyřčíslí udává adresu instrukce v programové paměti a druhé čtyřčíslí udává numerické kódování instrukce, pak bude do PC zkopírována adresa 0x1235 a SP bude dekrementován. Tímto je RET instrukce ukončena a v příštím instrukčím cyklu již bude vykonána instrukce BEQ (zde nekonenečná smyčka).

Bytů: 2

Cyklů: 1

Kódování:

1	X	X	X	X	X	X	X	X	Z	Y	1	0	0	1	1

Operace:

$$\text{PC} \leftarrow [\text{SP}]$$
$$\text{SP} \leftarrow \text{SP} - 1$$

DAB IMM (IMM)

Funkce: Cyklus podmíněného větvení

Popis: DAB je komplexní instrukce umožňují snadnou tvorbu *for* cyklů známou z vysokoúrovňových programovacích jazyků. DAB sníží akumulátor o jedničku a provede skok na relativní adresu $A[6-0]$, případně $A[8-0]$ danou operandem pokud $ZF = 0$. Instrukce má rovněž volitelný operand v podobně datové adresy $B[1-0]$ zpětného zápisu, kdy je snížená hodnota akumulátoru automaticky zapsána do datové paměti. DAB je sekvenční, mnoha stavová instrukce v tom smyslu, že vykonává sekvenci již existujících instrukcí, které již byly definovány v textu výše. V závislosti na hodnotě W (Write Back Enable) je provedena jedna z následujících sekvencí:

- Pokud $W = 1$ (Zapnuto)

```
add     #4094           ; -1
addcy   #1              ; One's complement
sto     #0000000000BB   ; Write-back B[1-0]
beq     .A0000AAAAAAA   ; Skok na +/-A[6-0]
```

- Pokud $W = 0$ (Vypnuto)

```
add     #4094           ; -1
addcy   #1              ; One's complement
beq     .A00AAAAAAAAA   ; Skok na +/-A[8-0]
```

Povšimněme si, že pokud $W = 0$ pak platí $AA = BB$. Vypnutí zpětného zápisu umožňuje větší rozsah skokové adresy +/-512 oproti původním +/-127. Addresa BB je dvoubitová a adresuje tak $C[3-0]$ datové paměti. DAB instrukce je perfektní ukázkou *recyklace* datové cesty, jak bylo vyloženo v oddílu 2.3 druhé kapitoly.

Příklad: Akumulátor obsahuje hodnotu 0x020. Instrukce,

```
.loop: DAB .loop, #0
```

provede 32 skoků na adresu .loop, snížená hodnota akumulátoru je průběžně zapisována na adresu #0 datové paměti. Toto je užitečné v případě, že je s hodnotou akumulátoru uvnitř cyklu manipulováno.

Bytů: 2

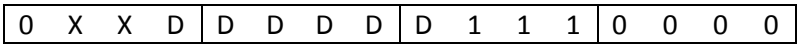
Cyklů: 7

Kódování:

A	A	A	A	A	A	A	A	B	B	W	1	0	1	1	0

Operace: Viz sekvence. Operace individuálních instrukcí k dispozici výše.

CMS IMM

Funkce:	Volba programového segmentu
Popis:	CMS zapíše danou hodnotou do segmentového registru. Programový segment je nutný k výpočtu absolutní adresy instrukce CALL. Po resetu zařízení je obsah programového segmentačního registru nedefinován a proto je pro správnou funkčnost CALL instrukcí nutné minimálně jednou CMS instrukci vykonat.
Příklad:	Instrukce, CMS #0 nastaví programový segment na hodnotu 0. Platný rozsah absolutních adres bude nyní 0x0000 – 0x0800 (0 – 2048). Podrobnější příklad viz instrukce CALL výše.
Bytů:	2
Cyklů:	1
Kódování:	 Platí pro $Z = Y$.
Operace:	Segment ← Immediate data

Poznámka: Z úsporných důvodů je pro zakódování několika dalších instrukcí využito restrikce $Y = Z = 1$. Tato kombinace kondicionálních kódu je normálně neplatná, neboť zde dochází k vzájemné kontradikci. Pokud $Z = 1$ akumulátor je vynulován a přirozeně tak musí platit, že $Y = 0$. Obdobně, pokud $Y = 1$ tak by mělo zároveň platit $Z = 0$. CF[27] je ovlivněna pouze součtem, negací a dělením. Negace zničí CF a principiálně nemůže způsobit vynulování akumulátoru. Nulový podíl značí nesoudělnost celočíselných operandů, tedy REM a ZF budou nastaveny. Při určení EXE (Execute) signálu je však s REM nakládáno jako s CF (jde o sdílenou vlajku o jejímž významu rozhoduje kontext, viz oddíl x.y) a uživatelský program by si tohoto jevu měl býti vědom.
Platí tedy, že nastavení CF z důvodu aritmetického přetečení (*arithmetical overflow*) způsobí také nastavení ZF, neboť CF se při výpočtu ZF v současné implementaci neuplatňuje. Toto je rozdíl mezi mezi skutečně nulovou hodnotou akumulátoru a hodnotou vyprázdněnou (*truncated*).

Pro účely podmíněného vykonávání je doporučeno, aby překladač takto instrukci nekódoval. Pokud položíme $Y = Z = 1$ bude daná instrukce (pokud umožňuje podmíněné vykonávání) jednoduše vykonána pokud je alespoň jedna z vlajek nastavena. Mnemonické označení pro $Z = Y = 1$ však neexistuje. Konečně, instukce využívající této restrikce nebudou nijak ovlivněny.

[27] Připomeňme, že Y a Z je označení pro CC (*Condition Code*), zatímco CF a ZF jsou odpovídající vlajky ve stavovém FLG registru.

Kapitola 5

Simulace a ověření návrhu

Celková funkčnost mikropočítače byla ověřena na testovací FPGA desce *Nexys™ 3 Spartan-6* (XC6SLX16). Hodinový signál CLK (Clock) je zde řízen 100 MHz oscilátorem na technologii CMOS. Následuje ukázka vzorového programu a též vybraných simulací. V závěru kapitoly lze rovněž nalézt přehled výsledků syntézy.

5.1 Vzorový program

V příkladu níže zastává mikropočítač funkci jednoduchých hodin ve 24 hodinovém formátu. Vzhledem k tomu, že prioritou práce není vývoj ukázkových aplikací, byla přijata jistá zjednodušení. Zde implementované hodiny skutečně pouze odměřují čas, výchozí minutu a hodinu nelze nastavit.
Výstup je průběžně zobrazován na 7-segmentovém displeji o čtyřech číslicích. Použitý displej je v konfiguraci společné anody, kdy jsou všechny kladné elektrody jednotlivých LED diod společně propojeny. Tímto způsobem je celá číslice displeje spínána jako celek. Zbylé katody LED diod jsou vyvedeny samostatně a tvoří tak individuální segmenty číslice. Vzájemně odpovídající segmenty (katody) jsou rovněž společně propojeny. Takto dostáváme kaskádové zapojení se čtyřmi vstupy pro celkové zapínání a vypínání číslic ANx (Enable Anode x) a sedmi vstupy CA − CG (Common A − Common G) pro zapínání a vypínán individuálních segmentů společných čislic.

Periférie je k mikropočítači připojena dle tabulky níže.

IO : Input/Output Port

MSB LSB

NC	AN3	AN2	AN1	AN0	A	B	C	D	E	F	G

A-G	IO.00-06	W	**Cathode.** Individuální segmenty společné pro všechny číslice displeje.
AN0-A	IO.07-10	W	**Common Anode.** Zapíná nebo vypíná celou číslici.
NC	IO.11	NI	**Not Connected.** Nepřipojeno, čteno a zapisováno jako nula.

Tab. 39: Zapojení displeje

Je patrné, že takto lze v jednu chvíli osvětlit pouze jednu číslici displeje. K zobrazení všech čtyřech číslic najednou je nutné displej v učitých intervalech průběžně obnovovat. Tento interval musí být dostatečné krátký na to, aby lidský zrak nestihl postřehnout rozsvěcení a zhášení jednolivých číslic a zároveň dostateně dlouhý na to, aby se neprojevila vysoká kapacita diody při jejím vypínání. Tomuto jevu se volně říká *ghosting* a projevuje se jako rozmazané osvětlení celé číslice, kdy se jednotlivé LED diody nestačí dostatečně rychle zotavovat. Výrobce doporučuje hodnoty obnovovací frekvence od 1 kHz do 16 kHz, zde jsme volili 1 kHz. Takto je každá číslice osvětlena na 1/4 celkové doby. Navíc jsou LED diody spínány v zapojení *open-drain*, tzn. že k iluminaci segmentu nebo zapnutí anody je nutno na výstup zapsat hodnotu log.0. Více informací o testovací desce lze nalést v oficiální příručce [6].

```
0000 0000                       ORG 0
0000 0000          ; Initialize Memory
0000 0070                       cms             #0
0001 BF70                       chhp            #0x3F           ; IO[11-6] output
0002 FF70                       chlp            #0x3F           ; IO[5-0]   output
0003 0004                       mov             a, #0
0004 8141                       sto             #mem_hours
0005 8151                       sto             #mem_mins
0006 8191                       sto             #mem_secs
                   ;  Timer Reset
0007 FFB1                       sto             #TIM_CMPL
0008 FFA1                       sto             #TIM_CMPM
0009 FF91                       sto             #TIM_CMPH
                   ; Program Mainloop
000A 0FF4          .mainloop:   mov             a, #0x0FF
000B FFB1                       sto             #TIM_CMPL
000C F5E4                       mov             a, #0xF5E
000D FFA1                       sto             #TIM_CMPM
000E 0054                       mov             a, #0x005
000F FF91                       sto             #TIM_CMPH
0010 0633          .meassure:   call            refresh_display ; disp. cascaded
0011 7FC1                       lod             #TIM_STS
0012 001A                       and             #1              ; DONe = ?
0013 804F                       beq             .meassure

0014 0004                       mov             a, #0
0015 FFB1                       sto             #TIM_CMPL
0016 FFA1                       sto             #TIM_CMPM
0017 FF91                       sto             #TIM_CMPH

0018 019C                       mov             b, #mem_secs
0019 0015                       lod             b
001A FC58                       add             #0xFC5          ; -59?
001B 8055                       storeq          b               ; zeroize secs
001C 004F                       beq             .inc_mins
001D 0191                       lod             #mem_secs
```

```
001E 0018                   add         #1
001F 8191                   sto         #mem_secs       ; incremented
0020 8177                   b           .mainloop

0021 015C   .inc_mins:      mov         b, #mem_mins
0022 0015                   lod         b
0023 FA78                   add         #0xFA0          ; -0x60?
0024 8055                   storeq      b               ; zeroize mins
0025 004F                   beq         .inc_hours
0026 0151                   lod         #mem_mins
0027 0FF3                   call        bcd_increment
0028 8151                   sto         #mem_mins       ; incremented
0029 8207                   b           .mainloop

0030 014C   .inc_hours:     mov         b, #mem_hours
0031 0015                   lod         b
0032 FDD8                   add         #0xFDD          ; -0x23
0033 8055                   storeq      b               ; zeroize hours
0034 80EF                   beq         .inc_mins
0035 0141                   lod         #mem_hours
0036 0FF3                   call        bcd_increment
0037 8141                   sto         #mem_hours      ; incremented
0038 8297                   b           .mainloop

             ;Refresh Display  Input: C[20],C[21]
             ;                 Output: None
0039 0141   .refresh_display: lod       #mem_hours
003A 8410                   slr         #4
003B 007C                   mov         b, #7
003C 08F3                   call        .display_digit  ; AN3
003D 0F13                   call        .delay          ; 1 ms
003E 0141                   lod         #mem_hours
003F 00FA                   and         #0x00F
0040 00BC                   mov         b, #11
0041 08F3                   call        .display_digit  ; AN2
0042 0F13                   call        .delay          ; 1 ms
0043 0151                   lod         #mem_mins
0044 8410                   slr         #4
0045 00DC                   mov         b, #13
0046 08F3                   call        .display_digit  ; AN1
0047 0F13                   call        .delay          ; 1 ms
0048 0151                   lod         #mem_mins
0049 00FA                   and         #0x00F
004A 00EC                   mov         b, #14
004B 08F3                   call        .display_digit  ; AN0
004C 0F13                   call        .delay          ; 1 ms
004D 8013                   ret
```

```
                ; Display Digit     Input: A digit, B position (one hot)
                ;                   Output: None
004E 8161       .display_digit:     sto     #temp_digit
004F 0005                           mov     a, b
0050 8171                           sto     #temp_anode
0051 0161                           lod     #temp_digit

0052 FFFA                           and     #0xFFF          ; zero?
0053 001C                           mov     b, #0x001
0054 025F                           beq     .enable_anode

0055 0161                           lod     #temp_digit
0056 FFF8                           add     #0xFFF          ; -1
0057 04FC                           mov     b, #0x04F
0058 021F                           beq     .enable_anode   ; one?

0059 0161                           lod     #temp_digit
005A FFE8                           add     #0xFFE          ; -2
005B 012C                           mov     b, #0x012
005C 01DF                           beq     .enable_anode   ; two?

005D 0161                           lod     #temp_digit
005E FFD8                           add     #0xFFD          ; -3
005F 006C                           mov     b, #0x006
0060 019F                           beq     .enable_anode   ; three?

0061 0161                           lod     #temp_digit
0062 FFC8                           add     #0xFFC          ; -4
0063 04CC                           mov     b, #0x04C
0064 015F                           beq     .enable_anode   ; four?

0065 0161                           lod     #temp_digit
0066 FFB8                           add     #0xFFB          ; -5
0067 024C                           mov     b, #0x024
0068 011F                           beq     .enable_anode   ; five?

0069 0161                           lod     #temp_digit
006A FFA8                           add     #0xFFA          ; -6
006B 020C                           mov     b, #0x020
006C 00DF                           beq     .enable_anode   ; six?

006D 0161                           lod     #temp_digit
006E FF98                           add     #0xFF9          ; -7
006F 00FC                           mov     b, #0x00F
0070 009F                           beq     .enable_anode   ; seven?

0071 0161                           lod     #temp_digit
0072 FF88                           add     #0xFF8          ; -8
0073 000C                           mov     b, #0x000
0074 005F                           beq     .enable_anode   ; eight?
```

```
0075 0161                lod             #temp_digit
0076 FF78                add             #0xFF7          ; -9
0077 004C                mov             b, #0x004
0078 001F                beq             .enable_anode   ; nine?

0079 030C                mov             b, #0x7B0       ; E (Error)
007A 0171    .enable_anode:  lod         #temp_anode
007B 0710                sll             #7
007C 0003                or              b               ; enable ANx
007D 0006                pio             io
007E 8013                ret
             ; Delay 1 ms    Input: None
             ;               Output: None
007F 003C    .delay:         mov         a, #3           ; 2nd iter, C[0]
0080 8001                    sto         #0
0081 FFF4    .delay_iter2:   mov         a, #0xFFF
0082 8096    .delay_iter1:   dab         .delay_iter1
0083 0001                    lod         #0
0084 9036                    dab         .delay_iter2, #0
0085 8013                    ret
             ; BCD Increment Input: A BCD number
             ;               Output: A incremented BCD number
0086 0018    .bcd_increment: add         #1
0087 8181                    sto         #temp_a
0088 00FA                    and         #0x0F
0089 FF68                    add         #0xFF6          ; -10
008A 0181                    lod         #temp_a
008B 001F                    beq         .bcd_carry
008C 8013                    ret
008D 0068    .bcd_carry:     add         #6
008E 8013                    ret
```

Tab. 40: Ukázkový program

Poznámka: Program odměřuje čas po sekundách. Uvažujeme-li frekvenci hodinového signálu 100 MHz, je nutno nastavit CMPx (Compare) na hodnotu 100 000 000 – 1 (0x5F5E0FF). Minuty a hodiny jsou kódovány v BCD[28] formátu a na displeji jsou zobrazovány v pořádí hodiny (AN3, AN2) a minuty (AN1, AN0), zatímco vteřiny jsou kódovany binárně. Číslo v BCD formátu inkrementujeme následovně:

```
VARIABLE bcd_numer;
...

bcd_number = bcd_number + 1;
IF ((bcd_number AND 15) => 10) THEN
    bcd_number = bcd_number + 6;
END IF;
```

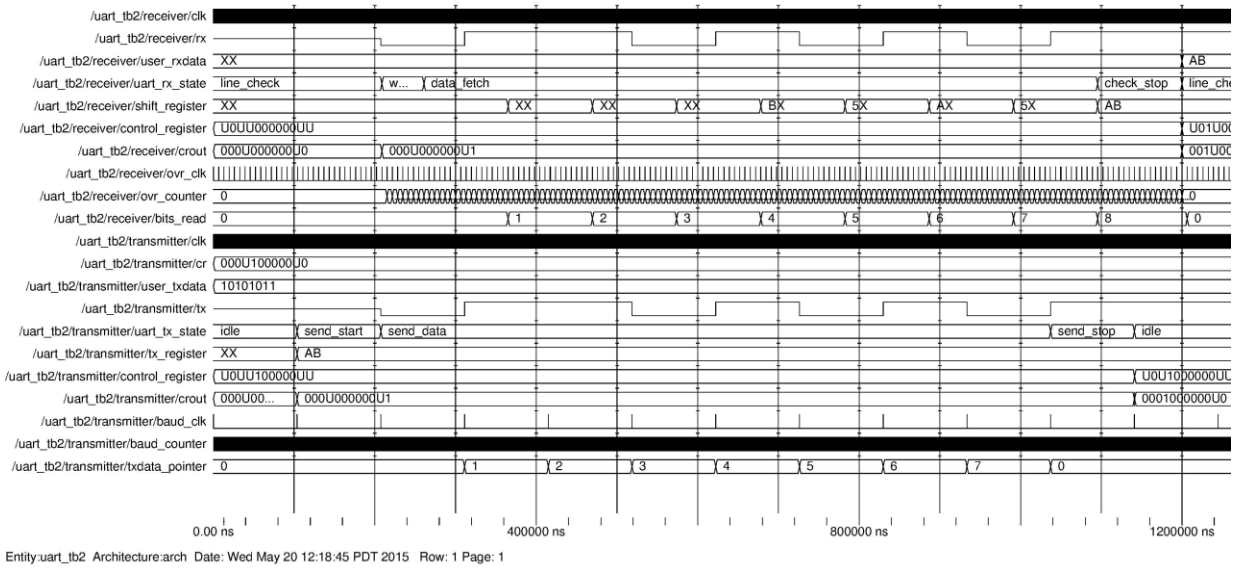
Tab. 41: Inkrementace BCD (pseudokód)

[28] BCD – Binary Coded Decimal

Procedura *.delay* odměřuje prodlevu 1 ms pomocí daného počtu vykonaných instrukcí. Totální přesnost je zde irelevantní – dopustíme se zde systematické chyby, kdy pro odečet jedné instrukce vykonáme sérii dalších instrukcí. Rovněž se dopouštíme chyby při zaokrouhlení iterací na celé číslo. Pro prosté generování obnovovací frekvence displeje je však tento přibližný postup plně dostačující.

5.2 UART simulace

Zde zachycená simulace byla provedena na úrovni vlastní entity. Ačkoliv implementovaný UART modul umožňuje otestování komunikace ve smyčce (viz UCR.LOP), pro účely názorného ověření funkčnosti je výhodnější prosté vzájemné zapojení přijímacího a vysílacího modulu. Výstup Tx (Transmit) vysílače je tedy připojen na vstup přijímače Rx (Receive). Jedná se tedy o analogické Loopback zapojení, zde však přijímač a vysílač vystupují separátně. Takovýto přístup umožňuje názornější (a jednodušší)inspekci všech signálů přesně jako v případě reálného přenosu.



Obr. 10: UART simulace

Na obrázku výše je zachycen průběh vyslání a přijmutí jednoho rámce formátu 8N1[29]. Zaměřme se nejprve na vysílací část simulace, tzn. signály instance entity *transmitter*.
Hlavní hodinový signál CLK (Clock) je velmi vysoké frekvence a vzhledem k nízkému časovému rozlišení sejmutého průběhu se jednotlivé hrany překrývají. K účelu synchonizace vysílání symbolů je od tohoto signálu odvozen signál mnohem menší frekvence, a to BAUD_CLK (Baud Clock). Princip a význam toho signálu byl již probrán v oddílu 3.2 třetí kapitoly o integrovaných perifériích. Vysílaná data, tak jak jsou vložena na vstup USER_TXDATA (User Transmit Data) modulu programem mikropočítače, jsou po validaci zapsána do interního vysílacího registru TX_REGISTER (Transmit Register) a vysílač přechází

[29] Označní 8N1 – 8 data bitů, žádná parita a 1 stop bit.

do stavu send_start (Send Start Bit). Rychlý pohled na obsah kontrolního registru odhalí TRA = 1 (Transmit) a TXE = 0 (Transmiter Empty). S každým tikem signálu BAUD_CLK je na vysílací vodič Tx linky zapsán právě jeden bit z vysílacího registru. V následujícím kroku je rovněž zvýšen ukazatel TXDATA_POINTER (Transmit Data Pointer), který tak odpočítává zbývající bity v registru. Zapíšeme-li vstupní hodnotu AB_{16} binárně, tedy 10101011_b, spatříme, že data jsou, podle očekávání, vysílána v pořadí od LSB do MSB. Zde je namístě podotkout, že stavové signály UART_TX_STATE a UART_RX_STATE obsahují proti intiuici nikoli stav současný, ale budoucí. Ve stavu send_start je tak již naplánován stav send_data (Send Data) a proto se průběh signálu jeví jako posunutý. Konec vysílání je indikován vysláním stop bitu ve stavu send_stop (Send Stop) a návratem do klidového stavu idle (Idle). Nyní je nastavena vlajka TXE = 1, TRA resetována a vysílání může stejným způsobem začít nanovo (v průběhu není zobrazeno).

Z pohledu přijímače (instance entity *receiver*) je ve stavu LINE_CHECK(Line Check) linka průběžně skenována pro začátek rámce. Jakmile jsou detekována data na lince, je nastavena stavová vlajka BSY = 1 (Busy). Vídíme, že s každým tikem odvozeného signálu OVR_CLK (Ovesampled Clock) je navzorkována 1/16 bitu na přijímacím vodiči Rx. Přesné podrobnosti a postupy vzorkování byly již uvedeny ve zvláštním oddílu 3.3 třetí kapitoly o integrovaných periférиích.
Posuvný registr SHIFT_REGISTER (Shift Register) je postupně plněn deserializovánými daty. Konec přijímání je indikován příjmem stop bitu ve stavu check_stop (Check Stop) a návratem do klidového stavu line_check. V tuto chvíli je rovněž nastavena vlajka RXF = 1 (Receiver Full) a je resetována BSY = 0. Výsledná data mohou být nyní přečtena z výstupu USER_RXDATA (User Receiver Data) a zařízení je připraveno k příjmu dalšího rámce.

Podrobné ověření funkčnosti návrhu mikropočítače jsou tímto dokončeny.

Poznámka: Vlastní program je nutno před syntézou uložit do programové paměti (soubor *program_memory.vhd* v příloze). Instrukce jsou do paměti zapsány deklarováním signálu rom, viz Tab. 3, kapitola druhá. Signál je typu *memory_t* , příklad deklarce rom signálu je v tabulce níže.

```
CONSTANT rom : memory_t (
    0 => instrukce1
    ...
    n => instrukcen );
```

Tab. 42: Rom signál

VHDL popis implementovaného mikropočítače je nezávislý na použitém FPGA, kód je univerzální. Pro cílová zařízení společnosti Xilinx Inc., lze využít například zdarma dostupný ISE Webpack. Pro účely práce byla použita konkrétní verze ISE Webpack 14.7 (nt), anglický jazyk. V prostředí jednoduše vytvoříme projekt a vložíme existující zdrojové soubory mikropočítače. Nyní stačí vybrat volbu *Configure Target Device* a vyčkáme až projekt projde všemi fázemi syntézy (*Synthetize – XST*, *Translate*, *Map*, *Place & Route* a *Generate Programming File*). Zde ještě podotkneme, že před započetím syntézy je nutno ještě spustit proces *Analyze Timing*/*Floorplan Design* (PlanAhead). Zde přidělíme signálním vstupům a výstupům mikropočítače (entita *mcu.vhd*) skutečné piny FPGA demo desky. Mapování jednotlivých pinů je uvedeno v oddílu 6.2 v celkových přehledech syntézy.
Po dokončení všech procesů by mělo dojít k automatickému spuštění programovací aplikace, kde stačí pomocí funkce *Boundary Scan* vybrat cílové zařízení. Pravým myšítkem vybereme čip, který chceme programovat, zvolíme programovací soubor (bitgen) a potvrdíme. FPGA bude překonfigurováno v souladu s HDL popisem a po resetu tak začne mikropočítač vykonávat zadaný program.

Kapitola 6

Závěr

Mikropočítač, popsaných v oddílech výše, je univerzální stroj schopný výkonu libovolně složitých výpočtů či algoritmů. Pokud zanedbáme ideální požadavek nekonečné paměti, můžeme prohlásit takový stroj za Turingovsky kompletní. Vskutku, mikropočítač obsahuje počáteční stav, (reálně konečnou) paměť, konečnou množinu stavů a konečnou množinu abecedních symbolů (instrukcí) a vstupů. Produktem aktuálního stavu a symbolu je pak stav nový či původní. Abeceda zahrnuje symboly pro změnu současného stavu (*posun pásky či čtecí hlavy*) a zápis symbolu do diskrétních paměťových buněk. Je zřejmé, že veškerou činnost univerzálního počítače lze zredukovat právě na tutu množinu elementárních operací. Navíc, pokud se omezíme na konečný výpočetní čas, budou i manipulovaná data konečná a restrikce nekonečné paměti reálného stroje ztratí na významu.

Ukázali jsme si tak, že funkční návrh takového výpočetního nástroje je poměrně přímočarý. Nejprve definujeme konečnou instrukční sadu (abecedu) počítače (viz oddíl 4.2 a dále). Složitost instrukční sady je volena s ohledem na požadovaný výpočetní problém, který bude stroj řešit. Stále však platí, že komplexnost a počet instrukcí nijak nerozšiřuje výpočetní možnosti systému, dokud jsou splněny všechny atributy stanovené výše. Turingovský model se tedy zabývá pochopitelně pouze prostou řešitelností algortimu, čas či efektivita samostného výpočtu je irelevatní. Pro reálný systém jsou však tyto aspekty více než důležité.

Datovou cestu (*datapath*) nelze přirozeně analyticky odvodit, nicméně vyplývá přímo z předepsané instrukční sady (viz kapitola 2). Datová cesta je rovněž, velmi opatrně řečeno, jakýmsi měřítkem efektivity výpočetního systému, neboť daná instrukce může být implementována nekonečným množstvím způsobů. Ukázali jsme si, že datová cesta obsahuje veškeré nezbytné kombinační a sekvenční komponenty, jako například sčítačky, násobičky, paměťové registry a podobně. Od datové cesty se lze pak ve vývojovém stromu plynule přesunout k návrhu řídící jednotky.

Řídící jednotka (viz oddíl 2.5 a dále) naopak vychází z velmi analytického postupu, neboť množina stavů stroje a kontrolních signálů přímo odráží struktura datové cesty. Přirozeně se tak nabízí řešení v podobě konečného deterministického stavového automatu. Volně můžeme postulovat, že obecně každý

deterministický proces operující na množině vstupů, stavů a výstupů je konečný deterministický stavový automat. Řídící jednotka a datová cesta jsou základními stavebními kameny každého sekvenčního výpočetního systému.

Neméně důležitou kapitolou ve vývoji mikropočítače je vlastní ověření teoretického návrhu. Jednotlivé komponenty mikropočítače musely být testovány průběžně a tudíž byly do paté kapitoly textu zahrnuty pouze konečné a aplikačně zajímavé příklady. Ukázkový program demonstruje naprostou většinu z existující palety instrukcí, a ty chybějící (jmenovitě DIV a MPY) byly patřičně odsimulovány. Program rovněž obsluhuje integrovanou periférii (časovač) a vstupně výstupní port. Z takto teoreticky a hlavně prakticky nabytých zkušeností můžeme prohlásit popisovaný mikropočítač za plně funkční.

V posledním oddílu níže budou alespoň ve zkratce uvedeny některé postřehy a rozšíření současné implementace.

6.1 Možná rozšíření

Implementované instrukce jsou citlivě vybrány tak, aby byla maximalizována efektivita výpočtů s příhlédnutím k jejich *ceně*. Cenou rozumíme množství faktorů, které je třeba při rozmisťování instrukcí zvážit. Cenou tudíž může být například počet bitů instrukčního slova a jeho celkový formát, neboť způsob jakým jsou jednotlivé instrukce zakódovány, výrazně ovlivňuje komplexnost dekódovací logiky. Toto se pak může ukázat jako obzvláště kritické z hlediska nároků na plochu čipu. Dále je nutno zvážit časovou náročnost dané instrukce a také její praktickou využitelnost (viz oddíl 4.1). Zejména tato kritéria pak vedla ke konečnému souboru instrukcí mikropočítače, tak jak jsou popsány v samostatné kapitole výše (viz oddíl 4.2).
Datová cesta popsaná v odílu x.y tak umožňuje realizaci i dalších instrukcí, které nebyly do existující sady instrukcí zahrnuty. Jejich stručný výčet lze nalézt v tabulce níže.

OP/Mnemonic	Operand	Description	FA	CE	Comp
ARITHMETIC AND LOGICAL OPERATIONS					
ADM	B/IMM	Add memory	Yes	Yes	ALU
NEGM	B/IMM	Negate memory	Yes	Yes	ALU
ORM	B/IMM	Bitwise OR memory	Yes	Yes	ALU
ANDM	B/IMM	Bitwise AND memory	Yes	Yes	ALU
SLLM	B/IMM	Shift logical left memory	No	Yes	ALU
SLRM	B/IMM	Shift logical right memory	No	Yes	ALU
MPYM	B/IMM	Multiply memory	Yes	Yes	ALU
DIVM	B/IMM	Divide memory	Yes	Yes	ALU
SQR	B/IMM	Square Accumulator	Yes	Yes	ALU
DATA TRANSFER OPERATIONS					
STOB	IMM	Store Aux B	No	Yes	DTU
LODB	IMM	Load Aux B	No	Yes	DTU
CDS	IMM	Change data segment	No	Yes	DTU

Tab. 43: Přehled možných instrukcí

Nové aritmeticko logické instrukce by tak mohly operovat přímo na datové paměti. Výpočetní výsledky jsou stále zapisovány zpět do akumulátoru, nicméně odpadá rutinní opakované čtení z paměti dedikovanou instrukcí LOD. Je zjevné, že cílový operand může být snadno měněn volbou write-back fáze a sice signály ENA (Enable Accumulator) nebo ENB (Enable Auxiliary B). Podobně lze rozšířit dvojici datových instrukcí STO a LOD, kde operand akumulátoru je nahrazen operandem registru B.
Paměť může být stále adresována tradičně pomocí okamžité adresy nebo registru B, ovšem s výjimkou datových instrukcí STOB a LODB, kde je toto ze zřejmých důvodů nesmyslné.
Rovněž lze elegantně rozšířit datovou paměť pomocí segmentování tak, jak je tomu u paměti programové. Připomeneme-li ve zkratce kódování CMS instrukce,

CMS IMM

Funkce:	Volba programového segmentu
Kódování:	0 X X D D D D D D 1 1 1 0 0 0 0 Platí pro $Z = Y$.
Operace:	Segment ← Immediate data

D	CMS.07-12	**Program segment.** Nastaví požadovaný programový segment 0 – 63.
X	CMS.13-14	**Don't care.** Čteno jako nula.

pak po drobné úpravě dostaneme:

CMS/CDS IMM

Funkce:	Volba programového/datové segmentu
Kódování:	0 M D D D D D D D 1 1 1 0 0 0 0 Platí pro $Z = Y$.

Operace:	if $(M = 0)$ then
	  Segment ← Immediate data
	else
	  Data Segment ← Immediate data

D	CMS.07-13	**Memory segment.** Nastaví požadovaný paměťový segment 0 – 127.
P	CMS.14	**Mode.** Pokud M = 0, je nastaven programový segment v souladu s instrukcí CMS, v opačném případě je podle CDS zvolen segment datový.

Adresování datové paměti se nyní bude řídit stejnými pravidly, které byly rozebrány v oddílu 4.4 instrukce CALL. Dosazením do rovnice absolutní hodnoty adresy s délkou datového segmentu 4096, maximální hodnotou segmentu 128 a velikostí individuální buňky 12 bitů vzroste množství adresovatelné datové paměti z původních 3 kB na 768 kB.

Podobnou úvahu lze aplikovat na délku celého instrukčního slova. Jelikož pozice bitů, které definují typ instrukce, jsou svojí pozicí pevně dány, je možné libovolně měnit (zvyšovat) délku instrukčního slova. Tímto způsobem poroste hloubka i šířka paměťového prostoru exponenciálně, stejně tak jako datová šířka registrů, či přímo, po jemných úpravách, i počet kódovatelných instrukcí. Je zjevné, že příliš dlouhá instrukční slova jsou na hranici praktické použitelnosti, nicméně už 32 bitové či 64 bitové instrukční formáty jsou poměrně běžné.

6.2 Výsledky syntézy

Logic Utilization	Used	Available	Utlilization
DEVICE UTILIZATION SUMMARY			
Number of Slice Registers	691	18224	3%
Number of Slice LUTs	2288	9112	25%
Number of fully used LUT-FF pairs	276	2684	10%
Number of bonded IOBs	15	232	6%
Number of BUFG/BUFGCTRLs	4	16	25%
Number of DSP48A1s	2	32	6%

Tab. 44: Využitá plocha cílového FPGA

Signál	Číslo pinu
MAPOVÁNÍ PINŮ – PINOUT	
IO<11> (IO)	K12
IO<10> (IO)	P17
IO<09> (IO)	P18
IO<08> (IO)	N15
IO<07> (IO)	N16
IO<06> (IO)	T17
RST (Reset)	C4
URX (UART Rx)	NC
CLK (Clock)	V10

Signál	Číslo pinu
MAPOVÁNÍ PINŮ – PINOUT (pokrač.)	
IO<05>	T18
IO<04>	U17
IO<03>	U18
IO<02>	M14
IO<01>	N14
IO<00>	L14
CA (Computer Alarm)	U16
UTX (Uart Tx)	NC
(Prázdné)	-

Tab. 45: Zapojení pinů cílového FPGA

Z přiložených výsledků syntézy si můžeme udělat představu o rozměrech celého návrhu. Vzhledem k nízkým nárokům na plochu čipu je možné paralelně implementovat i poměrně veliký počet soft-mikroprocesorů na jediném FPGA.

Vygenerovaný *netlist* je rovněž možné použít jako výchozí bod při implementaci zařízení na ASIC obvodech.V praktických aplikacích je tudíž možné návrh ověřit a odsimulovat právě s využitím jazyků hardwarového popisu (*FPGA prototyping*).

Bibliografie

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

[2] Pedroni, V. A., Finite State Machines in Hardware, MIT Press, 2013, ISBN 978-0262019668

[3] Šťastný J., Bílý P., Návrh mikrořadiče na FPGA, Elektrorevue, No. 30, 2006 [dostupné online: http://www.elektrorevue.cz/clanky/06030/index.html]

[4] Hwang Enoch O., Digital Logic and Microprocessor Design With VHDL

[5] Chu Pong P., FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version

[6] Digilent Inc., Nexys3™ Board Reference Manual, Revision: April 3, 2013 [dostupné online: https://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf]

[7] Xilinx Inc., Spartan-6 FPGA Configurable Logic Block, UG384 (v1.1) February 23, 2010

[8] Cummings Clifford E., Mills Don, Golson Steve, Asynchronous & Synchronous Reset Design Techniques – Part Deux [dostupné online: http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf]

[9] Blair-Smith Hugh, AGC4 MEMO # 9 – Block II Instructions

Příloha A

A.1 Procesní diagram dekódovací logiky