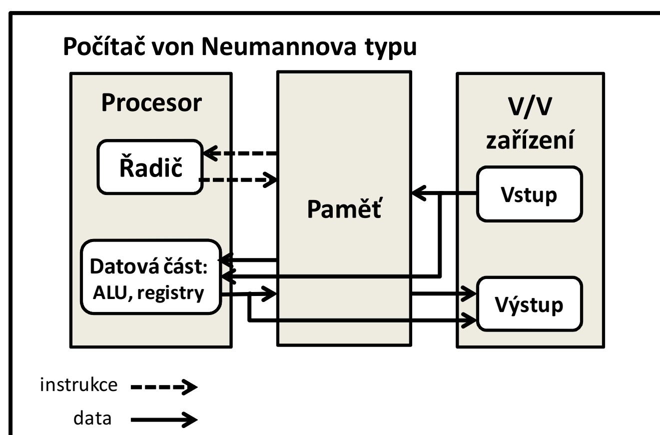
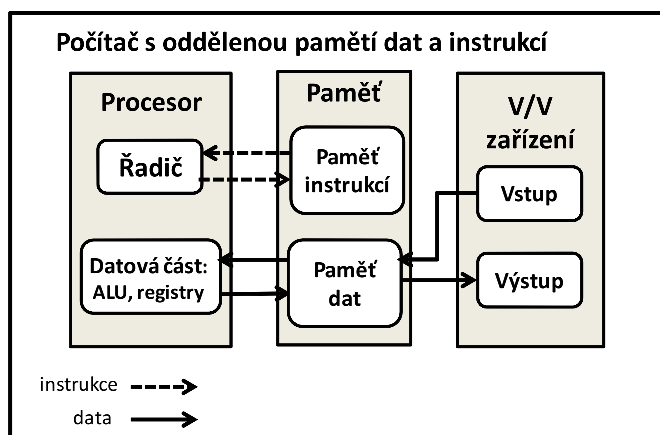
5. Struktura procesoru

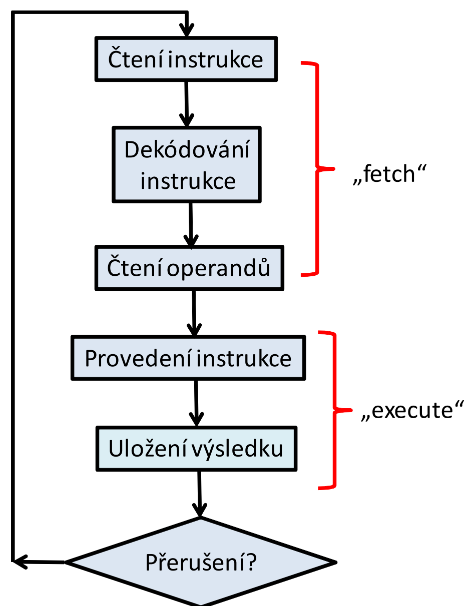
V kapitole 4. byly deklarovány základní vlastnosti číslicového počítače. Z tohoto popisu vyplývá, že podle výpisu paměti nelze poznat, zda jde o instrukce nebo o data (ani o jaká data), je třeba znát kontext. Počítač tvoří hlavní paměť (*main memory*), procesor obsahující datovou část (pracovní a adresové registry), aritmeticko-logickou jednotku (ALU) a řadič (*control unit, controller*) a vstupní/výstupní zařízení. Bloková struktura číslicového počítače podle von Neumannovy architektury je na obrázku 5.1. Existuje možnost (častá u malých tzv. jednočipových procesorů – počítačů, např. u mikropočítače AVR), že jsou použity dvě oddělené paměti, jedna pro instrukce a druhá pro data, které se neadresují společně. Takové architektuře se říká Harvardská, blokové schéma viz obrázek 5.2.



Obrázek 5.1.: Von Neumannova architektura číslicového počítače.

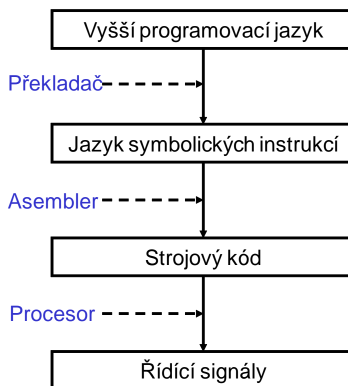


Obrázek 5.2.: Harvardská architektura číslicového počítače.



Obrázek 5.3.: Instrukční cyklus.

Počítač provádí instrukce, pohyb dat a instrukcí je na obrázcích znázorněn, pro jednoduchost nejsou zakresleny řídící signály, tzn. signály z řadiče, který řídí všechny jednotky. Řadič pracuje podle tzv. instrukčního cyklu, viz obrázek 5.3.



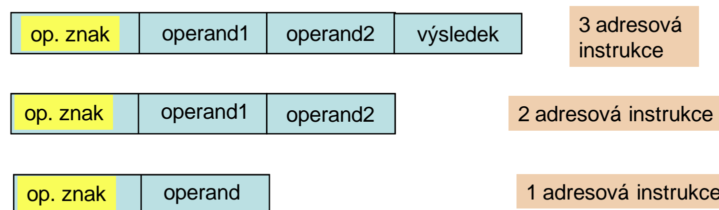
Obrázek 5.4.: Hierarchie programovacích jazyků a řízení počítače.

5.1. Architektury souboru instrukcí

Programy se píší ve vyšších programovacích jazycích, prostřednictvím hardwaru a překladů se z instrukcí musí odvodit řídící signály, viz obrázek 5.4. Mezi strojovým kódem a vyššími programovacími jazyky je jazyk symbolických instrukcí (JSI nebo též JSA: jazyk symbolických adres). Dnes není problém navrhnout a realizovat počítač – procesor pro konkrétní aplikaci (tzv. *application-specific procesor*, ASIP) se zohledněním požadavků na jeho efektivitu (velikost, výkon, spotřebu, rychlost, spolehlivost, ...). Návrhář musí vědět, jak vhodně zkombinovat technické a programové vybavení, co realizovat v hardwaru a co

programem. Pro návrhy tzv. vestavných (*embedded*) systémů nestačí znát jen jazyk C, ale musíme vědět, jak instrukce pracují, kdy, jak a kam ukládat a číst operandy, jaké použít sběrnice, jak pracovat s periferiemi atd. V následujícím textu se seznámíme se strukturou instrukcí, se způsoby adresace operandů, s konkrétními typy instrukcí, s architekturou souboru instrukcí a stručně si ukážeme, jak realizovat řadič procesoru. Paměťový subsystém bude obsahem následující kapitoly.

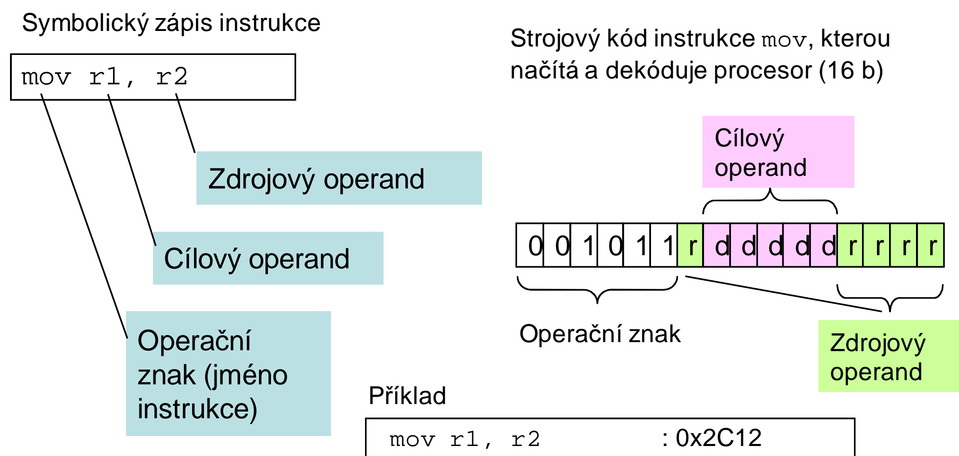
Instrukce je příkaz, zakódovaný jako číslo, který vyjadřuje, co se má provést (jaká operace), s čím se to má provést (operandy), kam se má uložit výsledek a kde se má pokračovat. Tyto informace jsou obsaženy v instrukci buď explicitně, např. v počítači SAPO, (5 adresový elektronkový číslicový počítač vyvinutý v ČR v 50. letech 20. století) nebo vyplývají z konkrétní architektury počítače implicitně (např. ve von Neumannově architektuře se pokračuje následující instrukcí). Instrukce se skládá z operačního znaku, který určuje, co se má provést, a z operandů. Operand je obvykle určen adresou do paměti. Mluvíme o jedno či více adresových instrukcích (i další pokračování v programu je dáno adresou), viz obrázek 5.5. Struktura instrukce má přímý vliv na to, jak je procesor konstruován: pokud je v instrukci jen jeden operand, je nutné, aby měl procesor ve své datové části alespoň jeden nebo více pracovních registrů. Je nutné efektivně realizovat instrukci „přesun“ mezi pamětí a registry.



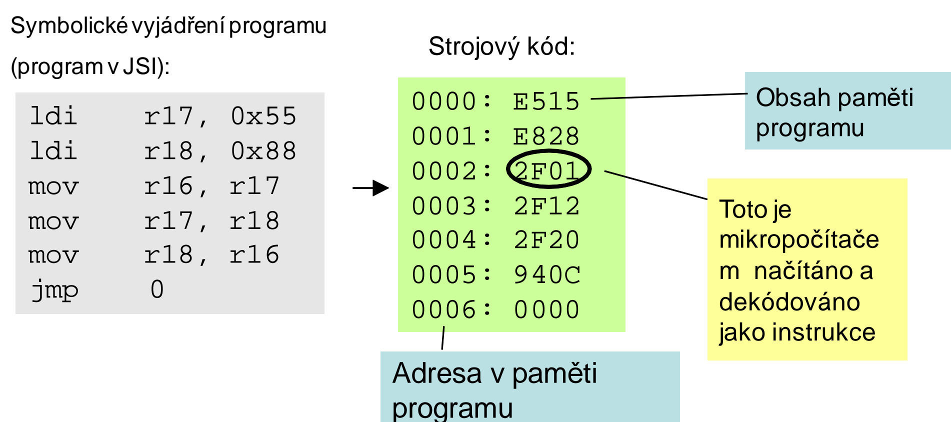
Obrázek 5.5.: Struktura instrukcí.

Konkrétní příklad zápisu ve strojovém kódu a zároveň v JSI je na obrázcích 5.6 a 5.7. Co je tedy třeba určit? Formát a kódování instrukcí, umístění operandů a výsledku, jak jsou operandy umístěny (v paměti, v registrech nebo jinde), který operand může být v paměti, typy dat a velikosti operandů (velikost registrů a šířka sběrnic je vždy omezená), které operace budou podporovány, jak bude realizován výběr další instrukce, jak realizovat skoky, větvení programu, volání podprogramů.

S adresací operandů přímo souvisí i to, jak budeme přistupovat k položkám v paměti, co bude nejmenší adresovatelná položka (slovo) a jak budou uloženy delší položky. Často je nejmenším adresovatelným objektem slabika, 8 bitů, 1 B (Byte), ale pokud aplikace vyžaduje slovo o velikosti např. 23 bitů. je možné navrhnout a realizovat procesor přesně na míru, tedy všechny jednotky i paměť realizovat jako 23bitové. Pokud je efektivnější využít standardní procesor nebo procesorové jádro, musíme vědět, jak se delší položky do paměti ukládají. Obvyklá je tzv. slabikově organizovaná paměť, kde nejmenší adresovatelná položka je slabika. Existují dvě možnosti: big-endian (na nižší adrese je uložená nejvyšší slabika) a little-endian (na nižší adrese je uložena nižší slabika).



Obrázek 5.6.: Příklad zápisu instrukce (MOV) v JSI a ve strojovém kódu procesoru AVR.

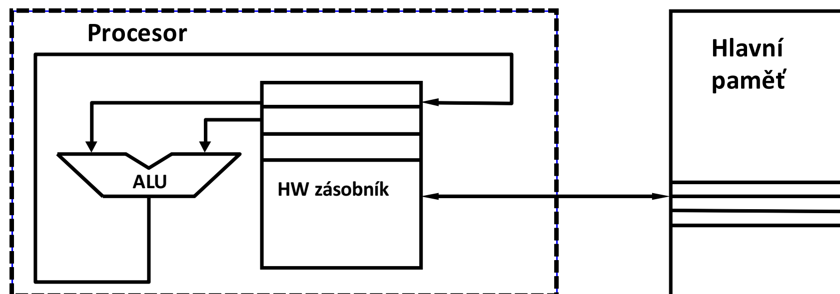


Obrázek 5.7.: Příklad zápisu programu v JSI a ve strojovém kódu procesoru AVR.

Rozlišujeme 3 různé architektury souboru instrukcí (*Instruction Set Architecture*), ISA:

- **Střadačově orientovaná ISA**: má jeden pracovní registr nazývaný střadač, krátké instrukce, a díky jednoduché vnitřní struktuře datových cest i jednoduché dekódování instrukcí, a tudíž jednoduchý hardware – ovšem za cenu časté komunikace s pamětí. Postupně vznikla potřeba doplnit další speciální registry pro nepřímou adresaci, zvláštním typem registru je tzv. stack pointer SP (ukazatel na vrchol zásobníku) a pracovní registry (tzv. zápisníková paměť) pro snížení četnosti přístupů do paměti. Implicitním operandem ALU je ale vždy střadač, druhý operand bývá obvykle v paměti.

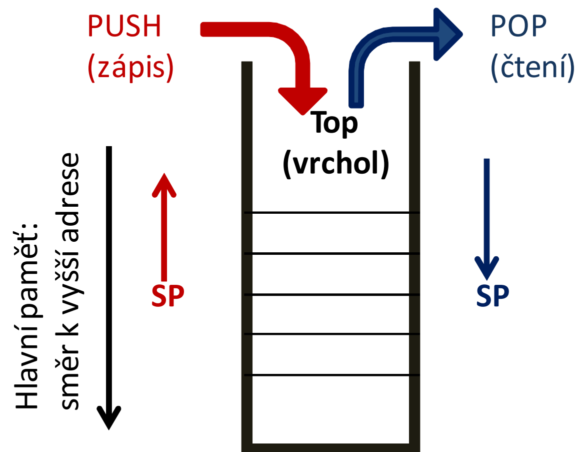
- **Zásobníkově orientovaná ISA**: pracovní registry jsou uspořádány do hardwarového zásobníku, ALU pracuje pouze s operandy uloženými na vrcholu tohoto zásobníku, viz obrázek 5.8. Výhodou jsou krátké instrukce, protože operandy jsou určeny implicitně (jsou v hardwarovém zásobníku), krátké programy a jednoduché dekódování instrukcí. Nevýhodou je, že nelze náhodně přistupovat k lokálním datům, protože zásobník je sekvenční, a nelze minimalizovat přístupy do paměti.



Obrázek 5.8.: Zásobníkově orientovaná ISA.

- ISA orientovaná na registry pro všeobecné použití, tzv. **GPR architektura** (*General Purpose Registers*). Výhodou je, že registry jsou rychlejší než paměť, lze k nim přistupovat náhodně (na rozdíl od zásobníku), mohou obsahovat mezivýsledky a lokální proměnné, což znamená i méně častý přístup do paměti. Nevýhodou je vždy omezený počet těchto registrů, složitější překladač, protože musí optimalizovat jejich použití, delší přepnutí kontextu, registry nemohou obsahovat složitější datové struktury a k datům v registrech nelze přistupovat přes ukazatele.

Z vybrané architektury souboru instrukcí, adresace operandů a z principů von Neumannovy architektury vyplývá, že procesor by měl ve své datové části obsahovat několik registrů, které mají zcela konkrétní funkci. Základním je **programový čítač**, PC („Program Counter"), který obsahuje adresu instrukce. Protože zpracování instrukce probíhá v několika krocích (viz obrázek 5.3), je v PC adresa právě zpracovávané instrukce jen na začátku cyklu (tedy při jejím čtení). Poté je obsah PC inkrementován, takže v dalších fázích zpracování instrukce PC obsahuje adresu následující instrukce. Hardwarová realizace by měla umožňovat paralelně zápis a čtení do/z paměti a zároveň přičtení jedničky k obsahu PC.



Obrázek 5.9.: Zásobník simulovaný v hlavní paměti, rostoucí směrem k nižším adresám.

Dalším registrem je **ukazatel zásobníku** SP („Stack Pointer“), který je nutný při volání podprogramů a zpracování přerušení, ale nejen tehdy. Hardwarová realizace by měla umožňovat (kromě zápisu a čtení) inkrementaci i dekrementaci SP, často i přičtení konstanty pro tzv. vyčištění zásobníku. Procesory se střadačovou a GPR architekturou mají zásobník simulovaný v hlavní paměti (na rozdíl od hardwarového zásobníku v zásobníkové ISA). Pro ukládání a výběr dat ze zásobníku se používají instrukce **POP** a **PUSH**, viz obrázek 5.9.

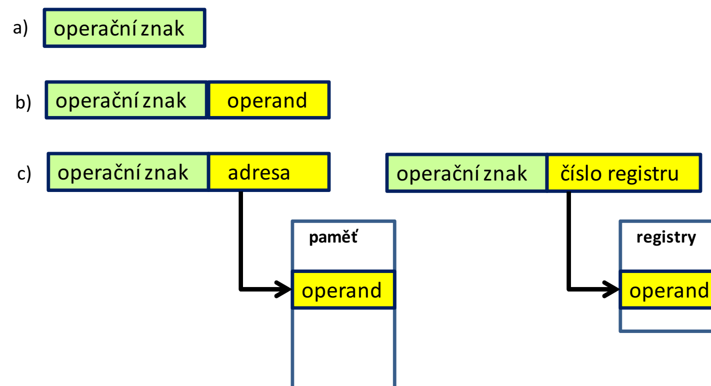
Pro větvení programů podle podmínek, je výhodné tyto podmínky tzv. příznaky (informace o výsledcích operací, tedy přetečení, přenosu, znaménku výsledku, apod.) uchovávat ve **stavovém registru** (SREG, FLAG, „flag register“). Hardwarová realizace tohoto registru musí umožňovat číst a měnit hodnoty jednotlivých bitů, a to nejen automaticky při zpracování aritmetických instrukcí, ale i konkrétními instrukcemi např. pro nastavení příznaku C (přenosu). Při programování v JSI je nutné vědět nejen jakou funkci instrukce realizuje, ale jak mění příznaky (viz příloha A2). Konkrétně v procesoru AVR jsou příznaky I (přerušení), T (krokování), H (přenos mezi 3. a 4. bitem), S ($N \oplus V$), V (přeplnění, overflow), N (záporný výsledek), Z (nulový výsledek) a C (přenos, carry).

Další registr, který je na rozdíl od předchozích programátorovi nepřístupný, ale je nutný pro zpracovávání instrukci podle instrukčního cyklu, viz obrázek 5.3, je registr instrukce (RI). Do RI se při čtení instrukce uloží právě přečtená instrukce. Ta se potom dekóduje (často postupně, kdy se jen podle konkrétních několika bitů pozná, zda má instrukce operandy a kolik, jak je dlouhá, zda se jedná o aritmetickou instrukci, bity určující druh operace se mohou přímo přivést jako řídící vstupy do ALU, apod.).

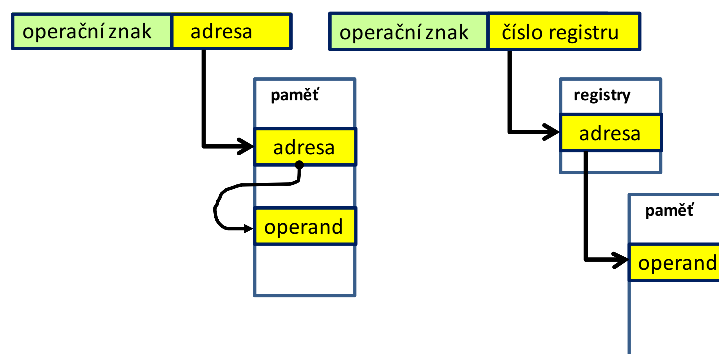
5.2. Adresace operandů

Jedna ze základních charakteristik číslicového počítače je, že v instrukci zpravidla není uváděna hodnota operandu, ale jeho adresa. V zásobníkové ISA je tato adresa daná implicitně – operandy jsou v hardwarovém zásobníku. Ve střadačové ISA postupně pro zefektivnění přístupů do paměti a hlavně pro práci se strukturovanými daty (pole, matice apod.) vznikla potřeba realizovat specializované registry pro adresaci.

Nejjednodušším případem je **přímá adresace**, viz obrázek 5.10. Operand je buď určen přímo operačním znakem (např. instrukce, která povoluje nebo zakazuje přerušení – to je realizováno nastavením jednoho bitu v příznakovém registru) nebo je v instrukci přímo obsažena konstanta nebo je operand určen adresou, a to buď adresou do hlavní paměti nebo adresou registru v registrovém poli. Počet registrů je vždy omezen, takže tuto „adresu“ (vlastně označení jednoho z více registrů) tvoří menší počet bitů než adresu hlavní paměti (například pro výběr z pole 32 registrů potřebujeme 5bitovou adresu), viz obrázek 5.6.

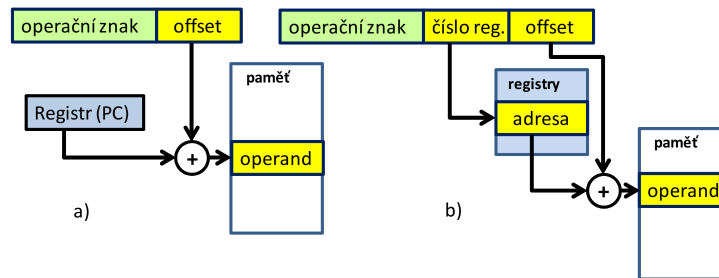


Obrázek 5.10.: Přímá adresace, kde operand je a) implicitní, b) přímá konstanta, c) přímá adresa.

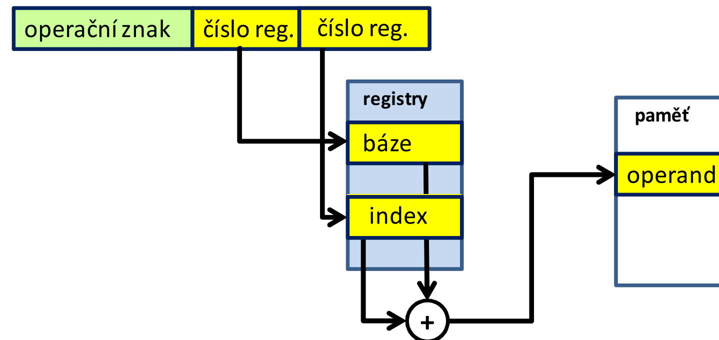


Obrázek 5.11.: Nepřímá adresace.

Aby se zkrátila instrukce, využila se adresace pomocí registrů, a přesto byl operand uložen v hlavní paměti, vznikla **adresace nepřímá**, viz obrázek 5.11. Obecně mohou být nepřímé adresy i vyšších řádů, kde je operand nalezen až po několika krocích.



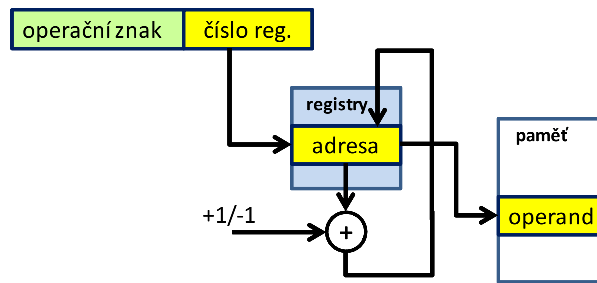
Obrázek 5.12.: Relativní adresace.



Obrázek 5.13.: Indexová adresace.

Při **relativní adresaci** je operand určen tzv. offsetem, který se přičítá k obsahu speciálního registru. Operand v instrukci je tedy relativní, viz obrázek 5.12. Tento registr podle konkrétní ISA bývá registr PC (u podmíněných skoků), viz 5.12a) nebo je možné použít libovolný registr z pole registrů (pak je operand složen z adresy registru v registrovém poli a z offsetu, viz 5.12b).

Pro efektivní práci s poli, která jsou zpravidla zpracovávána postupně položka po položce a v matici po řádcích, se používá **indexová adresace** často v kombinaci s relativní adresací, viz obrázek 5.13. Pro efektivní pohyb v paměti existuje adresace umožňující autoinkrementaci a autodekrementaci, viz obrázek 5.14, to znamená zvýšení nebo snížení obsahu registru (tedy vlastně adresy) o jedničku (nebo o 2, 4, 8 bitů podle délky operandů). K obsahu registru se přičte nebo odečte tato konstanta, a to buď před vyhledáním položky v paměti nebo až potom (např. post-inkrementace se používá při realizaci operací cyklu s položkami uloženými v poli). Autoinkrementace a autodekrementace se používá při práci se zásobníkem: post-dekrementace se realizuje až po vyhledání příslušné položky v paměti, kde se nejdříve realizuje uložení dat na zásobník (instrukcí PUSH) a potom snížení – dekrementace ukazatele zásobníku SP. SP ukazuje na vrchol tj. na první volnou pozici v zásobníku, kam je možné ukládat data. Naopak při vyzvednutí dat ze zásobníku (instrukcí POP) se obsah SP nejdříve zvýší – inkrementuje, takže po provedení instrukce POP ukazuje SP zase na vrchol. Předpokládáme, že zásobník roste směrem k nižším adresám, jako je to u procesoru AVR, viz příloha A.2. Některé počítače umožňují ještě adresaci tzv. *škálovatelnou* pro efektivní práci v polích s delšími položkami. To je realizováno násobením obsahu vybraného registru konstantou (1, 2, 4, 8). Adresace přímá nebo/i nepřímá registrová většinou umožňuje ještě přičíst k obsahu registru konstantu (*displacement*).



Obrázek 5.14.: Autoinkrementace a autodekrementace.

5.3. Jazyk symbolických instrukcí, typické instrukce

Procesor nedělá nic jiného než zpracovává instrukce podle instrukčního cyklu. Procesory jsou tedy charakterizovány tím, co „umí“, tedy souborem instrukcí. Jazyk symbolických instrukcí je typický pro každý procesor. Názvy instrukcí i instrukce samotné jsou si velmi podobné, ale vždy je třeba najít v dokumentaci pro každou funkci, co přesně dělá, jaké může mít operandy, zda a jak mění příznaky, kolik taktů trvá. Zde shrneme základní skupiny instrukcí a budeme se snažit abstrahovat od konkrétního typu procesoru (i když uvedené příklady budou vycházet z procesoru AVR používaného v laboratořích předmětu, popis instrukcí najdete v příloze A.2.). Instrukce jsou několika typů:

- Instrukce: existuje pro ni kód a umístění na určité adrese v paměti.
- Pseudoinstrukce: např. vyhrazení místa pro proměnné, deklarace proměnných.
- Direktiva: instrukce pro překladač.
- Makroinstrukce: konkrétní skupina instrukcí se na několik míst v programu doplní až při překladu.

Dál se budeme zabývat základními typy instrukcí, které dělíme na instrukce pro přesuny dat, aritmetické a logické instrukce, instrukce pro posuvy, instrukce pro změnu pořadí provádění instrukcí (skoky), podprogramy a přerušení.

5.3.1. Přesuny dat

Tyto instrukce přesouvají data mezi procesorem (registry) a pamětí. Používá se zkratka **MOV** (angl. „Move“). Instrukce má dva operandy určující **kam** se má **co** přesunout. Místo jedné instrukce se pro uložení dat z registru do paměti používá instrukce **ST** („store“, ulož) a pro nahrání dat z paměti do registru instrukce **LD** („load“, nahraj). Pro ukládání a výběr dat ze zásobníku se používají instrukce **POP** a **PUSH**, viz obrázek 5.9.

5.3.2. Aritmetické a logické instrukce

Aritmetické a logické operace můžeme rozdělit na binární (se dvěma operandy) a unární (s jedním operandem). Základní instrukce v každém procesoru jsou pro sčítání:

$$ADD \quad R_d,\ R_r \qquad R_d \leftarrow R_d + R_r$$

a sčítání s přenosem z nižšího řádu (tedy sčítání s Carry):

$$ADC \quad R_d,\ R_r \qquad R_d \leftarrow R_d + R_r + C$$

Podobně vypadá instrukce pro odčítání (SUB) a odčítání s výpůjčkou (SBC). Často je používaná instrukce CP (compare), která je vlastně odčítání, ale bez uložení výsledku:

$$CP \quad R_d,\ R_r \qquad R_d - R_r$$

Výsledek se neuloží, tzn. obsah registrů se nezmění, ale změní se příznaky podle výsledku odčítání. Podle nich lze větvit program (viz instrukce pro podmíněné skoky).

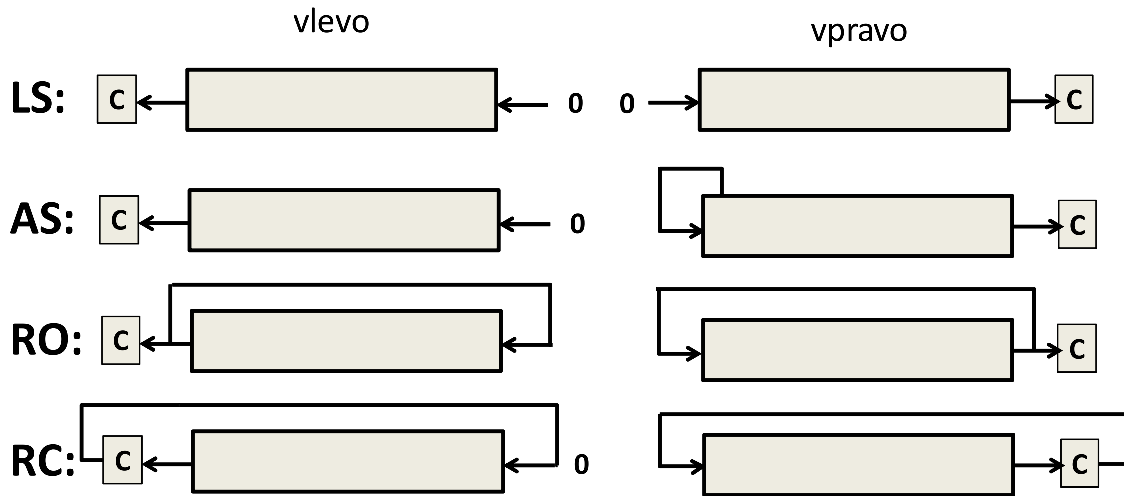
Z logických instrukcí bývají k dispozici instrukce pro logický součin (AND), součet (OR), nonekvivalenci XOR (EOR). Popis instrukcí procesoru AVR používaného v laboratoři najdete v příloze A.2 včetně toho, jak operace mění příznaky.

5.3.3. Posuvy

Instrukce pro posuv jsou na úrovni programování v JSI velmi často používané. Některé procesory nemají implementované instrukce pro násobení a dělení. Ty je pak možné realizovat pomocí sčítání (odčítání) a posuvů. Typy posuvů se znázorněním toho, co při konkrétním typu posuvu vypadává, kam se vypadlý bit ukládá a co se ukládá na volné místo, je znázorněno na obrázku 5.15. Levá část obrázku znázorňuje posuvy vlevo (L) a pravá vpravo (R), Význam zkratek je LS: logický posuv, AS: aritmetický posuv, RO: rotace, RC: rotace přes Carry flag. Některé procesory nemají v souboru instrukcí všechny typy posuvů, (např. AVR má jen rotaci přes C, označenou zkratkami ROL a ROR a ne RC).

5.3.4. Pseudoinstrukce

Vyhrazují místo v paměti (pro výsledky operací) a umožňují zadávat vstupní data. Obvykle se rozlišuje jednak velikost (BYTE – slabika, WORD – 16 bitů) a jednak čísla se znaménkem (SHORT). Obvyklá deklarace je DB, DW, DS (D jako definuj, deklaruj, „Def-Space“).



Obrázek 5.15.: Instrukce pro posuvy.

5.3.5. Skoky

Změna pořadí provádění instrukcí se realizuje pomocí instrukcí pro skoky, voláním podprogramů a přerušením. Nejjednodušším případem jsou skoky. Jde o instrukce, jejichž operandem je adresa, kde má program pokračovat. Tato adresa je v JSI určena návěštím, procesor konkrétní adresu získává až při překladu, a to buď tak, že ji ve strojovém kódu doplní absolutně nebo ji získá při provádění instrukce přičtením vzdálenosti od aktuálního obsahu programového čítače k cílovému místu skoku (obvykle v doplňkovém kódu, aby bylo možné skákat v programu dopředu i dozadu, viz obrázek 5.12). Používají se různé typy skoků, které pro zkrácení kódu omezují vzdálenost, kam je možné skákat, skoky nepodmíněné a podmíněné, které skok realizují jen v případě splnění nějaké podmínky (hodnoty příznaku nebo kombinace příznaků ve stavovém registru).

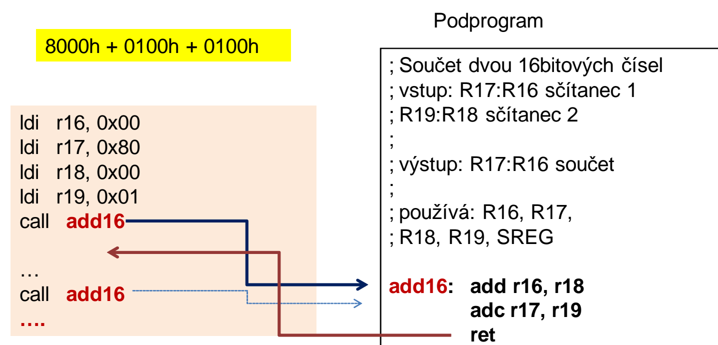
Nepodmíněný (absolutní) skok, nazývaný JMP, většinou dovoluje skákat v celém rozsahu paměti, naopak podmíněné skoky bývají „krátké", tzn. že jejich operand (vzdálenost adresy cíle skoku od adresy místa, kde je instrukce skok v programu) je např. jen 6 bitový (tedy se pohybuje ve slabikově organizované paměti programu jen v rozsahu 64 adres). Instrukce mívají v JSI název BRxy nebo Jxy, kde x je označení příznaku (např. C, carry) a y jeho hodnoty (např. S-set, když má hodnotu 1 nebo C-clear, když je nulový). Procesory mívají ve svém souboru instrukcí velké množství podmíněných skoků zaměřených na výsledek srovnání (po provedení instrukce CP-compare). Záleží na tom, zda pracujeme s nezápornými čísly nebo s čísly se znaménkem (v doplňkovém kódu). Podmíněné skoky jsou prováděny podle hodnot příznaků. Například skok BRLO je stejný jako BRCS, skočí se, jestliže je nastaven příznak C (Carry, přenos), viz tabulka 5.1.

Relace	Doplňkový kód	Nezáporná čísla
$<$	BRLT	BRLO
$=$	BREQ	BREQ
$\neq$	BRNE	BRNE
$\geq$	BRGE	BRSH

Tabulka 5.1.: Podmíněné skoky pro procesor AVR zaměřené na srovnávání. EQ (Equal, stejný), LT (Less, menší), GE (Greater or equal, větší nebo stejný), LO (Lower, pod), NE (not equal, různý), SH (Same or Higher, větší nebo rovno bez znaménka).

5.3.6. Podprogramy

Další možnost, jak změnit pořadí provádění instrukcí je volání podprogramu pomocí instrukce CALL k, kde *k* je adresa začátku podprogramu v paměti. Na rozdíl od skoků, je třeba realizovat možnost návratu z podprogramu a pokračování ve zpracovávání programu za příslušným voláním. Instrukce CALL tedy změní obsah programového čítače PC a na zásobník uloží návratovou adresu, tedy adresu následující instrukce za instrukcí CALL. Při návratu z podprogramu se pomocí instrukce RET nastaví obsah PC na adresu instrukce následující po CALL (odebere se ze zásobníku návratová adresa uložená při volání podprogramu a uloží se do PC). Příklad jednoduchého podprogramu, jeho volání a návrat je na obrázku 5.16.



Obrázek 5.16.: Volání podprogramu a návrat z podprogramu.

5.3.7. Přerušení a zpracování vstupů a výstupů

Přerušení („interrupt“) je v informatice metoda pro obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí v běžícím programu, vykoná tzv. obsluhu přerušení, a poté opět pokračuje v předchozí činnosti. Protože počítač zpracovává instrukce podle instrukčního cyklu, termín „zpracování přerušení“ znamená opět vykonání nějakého podprogramu, tedy posloupnosti instrukcí. Obsluha přerušení není tedy nic jiného než skok na místo, kde je program (podprogram) pro zpracování tohoto přerušení s tím, že podobně jako u volání podprogramu, je třeba se na konci programu pro zpracování přerušení vrátit k

právě běžícímu (přerušenému) programu. Protože se při obsluze přerušení mohou změnit příznaky, ukládá se při přerušení na zásobník kromě návratové adresy obvykle i obsah registru příznaků. Proto se musí použít jiná instrukce pro návrat z obsluhy přerušení (RETI), než pro návrat z podprogramu (instrukce RETI má ještě další funkce, např. informace pro řadič přerušení). Procesory mívají nejen vyhrazené místo v hlavní paměti, kam se skočí při zjištění konkrétního typu přerušení (jako AVR, konkrétně viz příklad 7), ale kokkrétní podprogramy pro zpracování určitých typů přerušení mohou být i součástí standardního programového vybavení (například u procesorů Intel obsluha klávesnice).

Přerušení může být vnější, vnitřní nebo softwarové.

Vnější (hardwarová) přerušení přicházejí od vstupních/výstupních (V/V) zařízení (tj. z pohledu procesoru přicházejí z vnějšku). Vnější přerušení jsou zpravidla do procesoru doručována prostřednictvím řadiče přerušení, což je specializovaný obvod, který umožňuje stanovit prioritu jednotlivým přerušením, rozdělovat je mezi různé procesory, vyvolat obsluhu přerušení podle zjištěného zdroje a případně realizovat další související akce.

Vnitřní přerušení vyvolává procesor, který tak reaguje na problémy při zpracování instrukcí a umožňuje operačnímu systému na tyto události nejvhodnějším způsobem zareagovat. Jedná se například o pokus dělení nulou, porušení ochrany paměti, výpadek stránky a podobně.

Softwarové přerušení je speciální instrukce, která je umístěna do prováděného programu a umožňuje např. vyvolat služby operačního systému z běžícího procesu (tzv. systémové volání).

Vstupy a výstupy zprostředkovávají kontakt s okolím a jejich realizace může být buď pomocí specializovaných a odděleně adresovaných registrů přístupných zpravidla pomocí instrukcí IN a OUT nebo mohou být mapovaná do paměti. Pak jsou přístupné pomocí běžných instrukcí pro zápis a čtení.

Konkrétní realizace a možnosti přerušení a konkrétní realizace vstupů a výstupů záleží na typu procesoru a vždy je třeba podrobně prostudovat dokumentaci. Správnou komunikaci zajišťuje ovladač zařízení, což je software, který umožňuje procesoru (operačnímu systému) pracovat s konkrétním hardwarem (například s tiskárnou nebo se síťovou kartou).

DMA („Direct Memory Access“, přímý přístup do paměti) je způsob přímého přenosu dat mezi hlavní pamětí a V/V zařízeními. Data neprocházejí přes procesor a lze tak dosáhnout vyššího výkonu (během DMA přenosu může procesor zpracovávat instrukce a data uložená v cache, viz kap. 6). DMA se používá pro přenos větších objemů dat (typicky dat z disku) a je v jistém smyslu odchylkou od von Neumannovy architektury počítače.

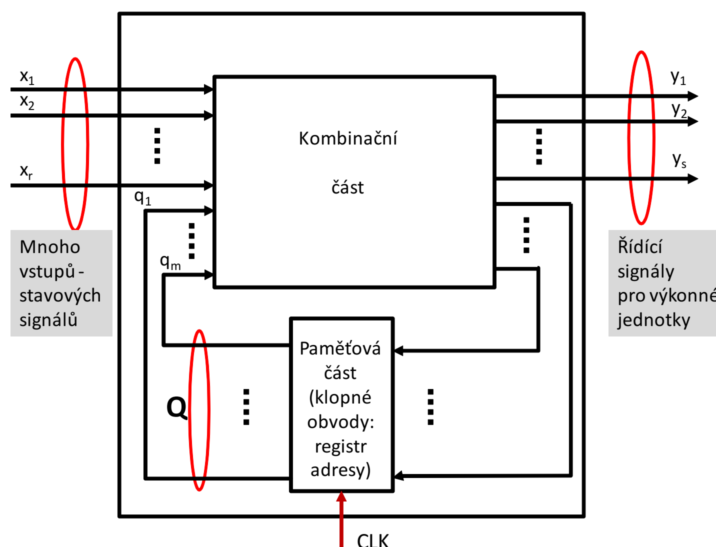
5.4. Řadič

Probrali jsme základní principy konstrukce jednotek číslicového počítače a metody, jak je navrhnout a realizovat, zbývá popsat způsob, jak to dát všechno dohromady a řídit celou činnost. Podle obrázku 5.1 tedy zbývá popsat funkci řadiče a vysvětlit, co to znamená sběrnice. Počítač (a tedy jeho řídící jednotka, řadič) pracuje podle instrukčního cyklu. Řadič řídí činnost všech výkonných jednotek počítače. Je to sekvenční obvod, protože závisí na posloupnosti vstupních signálů, které generují výkonné jednotky, kterým naopak vysílá řídící signály. Pracuje v nekonečném cyklu, řídí zpracovávání všech instrukcí. Navrhuje se podle instrukčního cyklu a konkrétně použitých hardwarových prostředků. Řadič je tedy klasický sekvenční obvod viz obrázek 3.1 a 5.17 s tím, že má velké množství vstupů (tedy informací z výkonných jednotek, kterým říkáme **stavové signály**), výstupů (**řídících signálů**) i vnitřních stavů. Ale v daném taktu (stavu) se uplatní jen málo vstupů, tzn. jeden nebo žádný, většinou se pokračuje následujícím stavem (graf přechodů popisující řadič má malé větvení). Místo vytváření grafu přechodů můžeme popsat činnost řadiče vývojovým diagramem, který vychází z instrukčního cyklu.

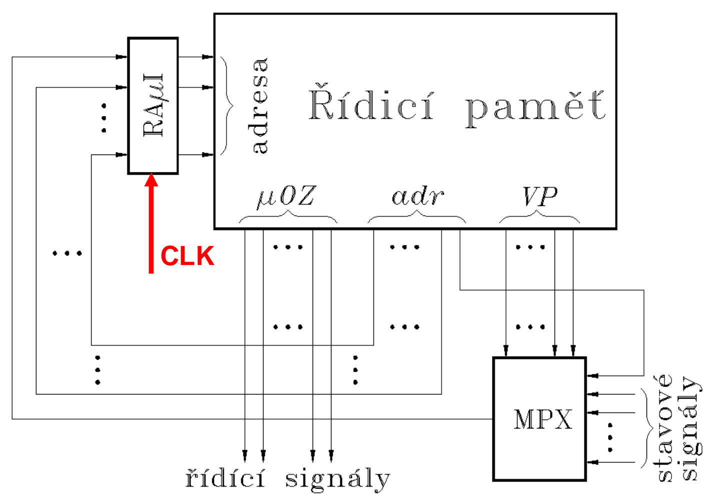
Podle způsobu realizace historicky dělíme řadiče na typ **obvodový** (klasický) a **mikroprogramový**. U obvodového řadiče jde o běžný návrh sekvenčního obvodu (velkého, s mnoha vstupy, výstupy a vnitřními stavy). Proto z důvodů snadnější a průhlednější realizace se využívá kód vnitřních stavů 1 z N. Pak lze návrh přímo odvodit z vývojových diagramů popisujících činnost procesoru při provádění instrukcí podle instrukčního cyklu. V současnosti lze použít návrhové systémy (EDA tools), které umožňují preferovat výběr kódu s ohledem na rychlost nebo plochu.

Mikroprogramový řadič je samozřejmě také sekvenční obvod s tím, že má svou kombinační část realizovanou pamětí (nazývá se řídící paměť, paměť mikroprogramů, *control memory*). Jednotlivé dílčí operace, které se provádějí při zpracování instrukcí, jsou uloženy v této paměti a říká se jim mikroinstrukce. Soubor mikroinstrukcí tvoří mikroprogram, soubor všech mikroprogramů je mikroprogramové vybavení – *firmware*. V řídící paměti je tedy uložen postup, jak zpracovávat všechny instrukce, které procesor „umí“. Protože se v daném stavu uplatní málo vstupů (také žádný, což znamená že se pokračuje na následující adrese v řídící paměti, viz obrázek 5.17), je výhodné přidat multiplexor a generovat řídící signály, které vyberou vstup, podle kterého se určí následný stav. Z důvodů synchronizace je vhodné realizovat řadič jako Mooreův automat. Struktura takového jednoduchého mikroprogramovaného řadiče je na obrázku 5.18. Adresa do řídící paměti se ukládá do registru adresy mikroinstrukce ($RA\mu I$). Srovnáním obrázků 5.17 a 5.18 zjistíme, že je to paměťová část sekvenčního obvodu, kam se ukládá vždy následný vnitřní stav. Výstupy z řídící paměti jsou (na rozdíl od obrázku 5.17) tvořeny třemi částmi: jednak blokem výstupních (řídících) signálů, kterým se u řadičů říká mikrooperační znak (μOZ), adresou (adr), která určuje následující vnitřní stav a výběr podmínky (VP). (VP) je tedy adresovým vstupem do přidaného multiplexoru (MPX) a vybírá, podle kterého vstupního (stavového) signálu se bude řadič větvit. Pokud se v daném stavu má pokračovat bez větvení, je vybrán ten vstup MPX, který tvoří nejnižší bit adresy. Toto je nejjednodušší možnost realizace mikroprogramového řadiče. Stavovými signály jsou i bity operačního znaku, který je v této realizaci třeba dekódovat postupně (nej-

jednodušší zkrácení postupného dekódování je použití multiplexoru MPX s více výstupy, který pak umožní větvení pro dva výstupy na 4 nebo pro tři výstupy na 8 adres). Existuje řada vylepšení, například přidání čítače adres proto, aby následující adresa nemusela být uložena v řídící paměti, nebo přidání hardwarového zásobníku, aby se daly snadno realizovat mikropodprogramy (protože řada instrukcí obsahuje opakované stejné části, například násobení).



Obrázek 5.17.: Model sekvenčního obvodu – řadiče.



Obrázek 5.18.: Mikroprogramový řadič.

Sběrnice je soubor vodičů a pravidel určený k propojení jednotek počítače. Dělíme ji na podsběrnice (obvykle též nazývané sběrnice):

- adresová (Address Bus),
- datová (Data Bus),

adresa	big-endian	little-endian
A01F	15	CF
A020	38	AB
A021	AB	38
A022	CF	15

Tabulka 5.2.: Uložení dat ve slabikově organizované paměti big-endian a little-endian.

- řídící . . . řídící a stavové signály (Control Bus).

Adresy a data mohou být časově multiplexovány, tzn. že stačí jediná sběrnice pro adresy i data, ale je třeba použít další řídící signál „adresa/data" pro rozlišení (jako má např. PCI). Dále je třeba zajistit, aby v daném okamžiku nebyla sběrnice řízena více jednotkami, tzn. aby nedocházelo ke kolizím při přenosu dat. Přidělování sběrnice může být **centralizované** (přidělovačem sběrnice je někdy přímo procesor) realizované buď jako cyklické výzvy (přidělovač „nabízí" postupně a adresně sběrnici jednotlivým jednotkám), paralelní přidělování (nezávislé, na základě žádostí a potvrzení), sériové přidělování nebo přidělování kombinované paralelní a sériové. **Distribuované** přidělování sběrnice je realizováno bez přidělovače tak, že je sběrnice buď postupně cyklicky přidělována všem jednotkám (round-robin) nebo se přiděluje na základě priorit (prioritní přidělování). Podrobněji viz předmět BI-JPO (Jednotky počítačů).

5.4.1. Příklady

1. Znázorněte, jak uložíte do slabikově organizované paměti dvojité slovo (DW: 32 bitů) o hodnotě $1538ABCF_{16}$ od adresy A01Fh.

   Řešení:
   Víme, že existují dvě možnosti: big-endian a little-endian podle typu procesoru. První způsob používají například procesory IBM 360, Motorola 68000, druhý způsob Intel 80x86, DEC Alpha. Obě možnosti jsou znázorněny v tabulce 5.2.

2. Napište část programu v JSI pro výpočet následujícího výrazu (předpokládejte čísla se znaménkem v doplňkovém kódu):

   $$R_{20} = (4 * R_{16} + 3 * R_{17} - R_{18})/8$$

   Použijte instrukce:
   LDI Ri, konst (nahrání konstanty do registru),
   LSL Rd nebo LSR Rd (logický posuv vlevo a vpravo),
   ASL Rd nebo ASR Rd(aritmetický posuv vlevo a vpravo),
   ADD Rd, Rr (ADC Rd, Rr) (sčítání a sčítání s přenosem, výsledek součtu Rd+Rr je v Rd),
   SUB, SBB (odčítání a odčítání s výpůjčkou a se stejnými operandy jako ADD),

a případně další instrukce. Konstanty je možné zadávat buď desítkově (tedy např. 255), šestnáctkově (0xFF) nebo dvojkově 0b1111 1111.

Řešení:
Násobení a dělení malými čísly (zde 4, 3, 8), je realizovatelné pomocí posuvů (viz kapitola 4). Násobení 4 pomocí dvou posuvů, násobení 3 pomocí posuvu o jedno místo a následným sčítáním, dělení 8 pomocí posuvu vpravo o 3 místa, tedy použijeme třikrát instrukci ASR. Výsledek je v registru R20. Jedna z možných verzí programu v JSI je následující (předpokládejme konstanty 5, 10 a 58, tedy výpočet výrazu $(4 * 5 + 3 * 10 - 58)/8 = -1$). V registru R20 bude po provedení programu v doplňkovém kódu –1 (FFh).

```
ldi  R16,  5
ldi  R17, 10
ldi  R18, 58
lsl  R16
lsl  R16
mov  R20, R17
lsl  R17
add  R17, R20
mov  R20, R16
add  R20, R17
sub  R20, R18
asr  R20
asr  R20
asr  R20
```

3. Napište část programu v JSI pro procesor typu AVR, který sečte dvě 32bitová čísla v doplňkovém kódu, uložené metodou „little endian“ v registrech R16 až R19 a R20 až R23. Výsledek uložte do registrů R20 až R23. Použijte instrukce jako v příkladu 1. Počáteční nastavení registrů (nahrání konstant) je následující:

```
ldi R16, 0x4
ldi R17, 0x10
ldi R18, 0x12
ldi R19, 0xFF
ldi R20, 0xFF
ldi R21, 0xFA
ldi R22, 0x88
ldi R23, 0x20
```

Kde je uložená informace o znaménku výsledku?

Řešení:
Procesor AVR má jen 8bitovou sčítačku, takže je nutné použít několik instrukcí (zde 4) pro sčítání s tím, že nesmíme zapomenout na možný přenos mezi slabikami. Použijeme tedy instrukci ADD pro nejnižší 8bitové sčítání a pak ADC. Informace o

přeplnění bude (jde podle zadání o čísla v doplňkovém kódu) uložena v příznaku V (overflow). Pokračování instrukcí:

```
add   R20, R16
adc   R21, R17
adc   R22, R18
adc   R23, R19
```

Výsledek sčítání pro konkrétní zadání sčítanců je:

```
  FF121004
+ 2088FAFF
-----------
  1F9B0B03
```

K přeplnění nedošlo (sčítala se čísla s různými znaménky) a v registrech bude:

```
R20: 03
R21: 0B
R22: 9B
R23: 1F
  V: 0
```

Informace o znaménku výsledku je v příznaku N (0).

4. Napište, co bude výsledkem následující části programu v JSI (určete výsledný obsah registrů R16 - R18):

```
         ldi   R16, 0x3A
         ldi   R17, 0x53
         ldi   R18, 3
navesti: lsl   R16
         rol   R17
         dec   R18
         brne  navesti
```

Význam instrukcí je popsán v příkladu 1, „dec Rd“ je dekrementace registru Rd, tedy snížení jeho obsahu o 1 a „BRNE navesti“ je podmíněný relativní skok v případě pokud není rovno, což po instrukci dec R18 indikuje nenulovost registru R18, tedy vstup do dalšího cyklu.

Řešení:
Instrukce LDI nahraje do registrů R16 a R17 konstanty, které jsou pak v cyklu logicky posouvány a rotovány, dokud obsah registru R18 není nulový. Nezapomínejme, že logický posuv probíhá tak, že vypadávající bit se nasouvá do příznaku C a rotace je 9bitová přes C. Výsledné obsazení registrů je tedy pro tento konkrétní případ:

```
R16 = D0
```

```
R17 = 99
r18 =  0
```

5. Navrhněte část programu v JSI, která bude realizovat tzv. čekací smyčku.

   Řešení:
   Záleží na tom, jak dlouho je třeba čekat. Záleží na pracovní frekvenci procesoru a na tom, kolik a jak dlouhých instrukcí ve smyčce použijeme (viz Příloha A.2). Pak zvolíme vhodnou konstantu, například 255 (0xFF). Možných řešení je vždy více, to s použitím nepodmíněného skoku BRNE je následující (smyčka končí, jestliže je R20 nulový):

```
        ldi R20, 255
cekej:  dec R20
        brne cekej
```

   Jiné řešení je použití podmíněného skoku CPSE Rd, Rr (porovnej a přeskoč následující instrukci, pokud je Rd rovno Rr):

```
        clr R16
        ldi R17, 100
cycle:  inc R16
        cpse R16, R17 ; compare and skip if equal
        jmp cycle
```

6. Navrhněte delší čekací smyčku než v předcházejícím příkladu.

   Řešení:
   Je třeba použít vnořené čekací smyčky nebo přerušení od časovače. Příklad řešení pro 255 opakování vnitřního cyklu a to celé zopakováno 20krát je následující. Délka v sekundách záleží na délce taktu procesoru tedy v našem případě je doba trvání čekací smyčky v taktech:

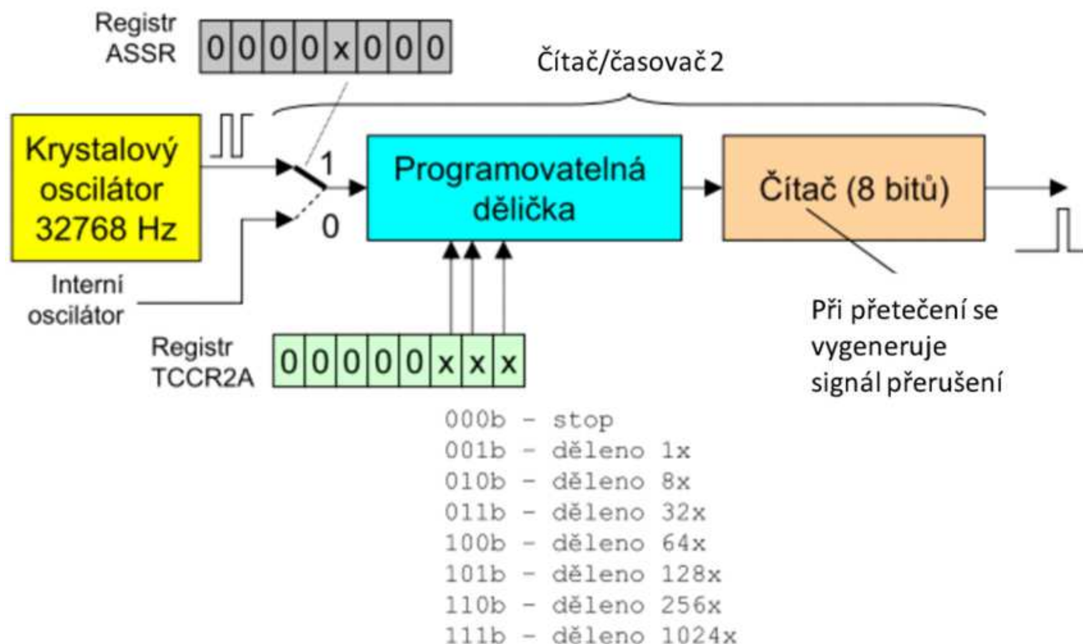
$$((255 \times 3takty) + 3takty) \times 20$$

```
        ldi R21, 20
cek2:   ldi R20, 255
cek:    dec R20    ;1 takt
        brne cek   ;1 takt (2 takty pro splněnou podmínku skoku)
        dec R21
        brne cek2
```

7. Navrhněte část programu, který umožní využít přerušení generované přetečením čítače/časovače.

   Řešení:
   Nyní se neobejdeme bez konkrétních informací o struktuře použitého procesoru. Jednočipový mikropočítač AVR má řadu V/V zařízení (periferií), které vysílají

procesoru žádosti o přerušení. Jedná se o přerušení vnější z hlediska procesoru, přestože jsou tyto jednotky na čipu. Žádosti o přerušení zpracovává řadič přerušení (u AVR prioritní dekodér, tedy kombinační obvod). Pokud je přerušení povoleno (instrukcí SEI), je podle zdroje přerušení přepsán obsah PC. To znamená, že procesor přeruší zpracovávání programu a skočí do vyhrazeného místa v paměti (pro AVR při přerušení od přetečení čítače/časovače na adresu 000Ah). Na tuto adresu umístíme instrukci skoku (JMP) na adresu pro obsluhu přerušení (zde podprogram s návěštím INT_T2). Obsluhu konkrétního přerušení je třeba napsat. AVR procesor při zpracování přerušení sám neukládá obsah stavového registru (SREG), proto je třeba obsah SREG uložit na zásobník pomocí instrukcí IN a PUSH a na konci zpracování přerušení jeho hodnoty zase obnovit (OUT a POP). V našem příkladu je jedinou funkcí obsluhy přerušení zvětšení obsahu registru R26 (základ pro návrh stopek, hlavní část programu je třeba dopsat). Struktura časovače pro AVR ATMEGA v přípravku AVR Butterfly je na obrázku 5.19.



Obrázek 5.19.: Princip časovače v procesoru AVR.

```
.include "m169def.inc"
.cseg
.org 0x0000  ; realizace RESETu procesoru
jmp start
.org 0x000A  ; umístění skoku JMP na adresu 000Ah
jmp int_T2
.org 0x0100
start:
    ldi r16, 0x00
    out SPL, r16
    ldi r16, 0x04
```

```
    out SPH, r16
...
    ; 32 768 za 1 sec
    ; 256 za 1/128 sec

    cli ; globální zakázání přerušení
    ldi r16, 0b00001000
    sts ASSR, r16 ; vyber hodiny od krystalového oscilátoru 32768 Hz
    ldi r16, 0b00000001
    sts TIMSK2, r16 ; povolení přerušení od časovače 2
    ldi r16, 0b00000011
    sts TCCR2A, r16 ; spuštění čítače s dělicím poměrem 32
...
    sei ; globální povolení přerušení

main_cykl:
...

    jmp main_cykl

int_T2: ; obsluha přerušení 1/128 sec
    push r16
    in r16, SREG  ; uložení SREG na zásobník
push r16
    inc r26
    pop r16  ; obnovení obsahu SREG
    out SREG, r16
    pop r16
reti  ; instrukce pro návrat z podprogramu na adresu
      ; uloženou v zásobníku a umožnění zpracování dalšího přerušení
```